A FORMAL FRAMEWORK FOR UNDERSTANDING RUN-TIME CHECKING ERRORS IN GRADUALLY TYPED LANGUAGES

by

FELIPE ANDRES BAÑADOS SCHWERTER
B.E.Sc., Universidad de Chile, 2011
M.Sc., Universidad de Chile, 2014

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)

April 2024

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the dissertation entitled:

A formal framework for understanding runtime checking errors in gradually typed languages

submitted by Felipe Andres Bañados Schwerter in partial fulfillment of the requirements for

the degree of Doctor of Philosophy

in Computer Science

Examining Committee:

Ronald Garcia, Associate Professor, Computer Science, UBC
Supervisor

William J. Bowman, Assistant Professor, Computer Science, UBC
Supervisory Committee Member

Reid Holmes, Professor, Computer Science, UBC
Supervisory Committee Member

Ali Mesbah, Professor, Electrical and Computer Engineering, UBC
University Examiner

Mark R. Greenstreet, Professor, Computer Science, UBC
University Examiner

David Van Horn, Associate Professor, Computer Science, University of Maryland
External Examiner

Additional Supervisory Committee Members:

Margo Seltzer, Professor, Computer Science, UBC
Supervisory Committee Member

ABSTRACT

Although Abstracting Gradual Typing provides a systematic approach to design gradual languages, the original framework has limitations: first, it accepts design choices that lead to type inconsistencies sneaking through evaluation. Second, when a type inconsistency is identified at run time, evaluation halts without providing any feedback on the parts of the program related to the failure, a safe approach yet unhelpful for debugging.

This dissertation addresses these two limitations of the Abstracting Gradual Typing framework. For the first limitation, I impose an extra constraint on the acceptable designs for gradual types: forward completeness of every type operation. This stricter constraint guarantees that, throughout evaluation, gradual types and runtime evidence objects cannot lose precision and will only represent information consistent with the original static type system. I introduce a new design for a gradual language with record subtyping that fulfills this restriction.

For the second limitation, I provide a specification for runtime program slicing that can be systematically applied to languages designed using Abstracting Gradual Typing. Slicing can separate the portions of a program that are guaranteed to be uninvolved in a runtime failure. Unlike the standard blame approach, slicing does not assume that types are correct. The slicing semantics can be used to provide a debugging tool, and I apply empirical research methods to explore whether this runtime type slicing approach is useful to developers.

LAY SUMMARY

Programmers make mistakes. One way to avoid mistakes is to describe and enforce the valid inputs and outputs for program parts using types, but this approach is tedious and restrictive.

"Abstracting Gradual Typing" is a design recipe for programming languages where programmers can choose where and whether to write types. This flexibility requires the language to check for mistakes as programs run.

This dissertation addresses two problems with this recipe. First, sometimes the run-time checks added by the recipe do not catch all the errors that they should. I extend the recipe with restrictions that avoid this problem.

Second, when programmers make a mistake and a run-time check fails, they receive no feedback about what went wrong. I provide a recipe to design error reporting systems that separate the parts of a program which might be the source of the error from the parts that definitely are not.

PREFACE

All the work presented henceforth was conducted in the Software Practices Laboratory at the University of British Columbia, Point Grey Campus. All projects and associated methods involving human participants were approved by the University of British Columbia's Research Ethics Board (certificate #H20-02068). No Large Language Models were involved in the production of this document.

A version of Chapter 3 was published in Bañados Schwerter, Clark, et al. (2021). The dissertation chapter includes considerable changes in manuscript composition from the published version. I was the lead investigator in the publication, responsible for major areas of concept formation, formalism, proof mechanization, and manuscript composition. Alison M. Clark identified examples highlighting problems with previous work. Khurram A. Jafery was involved in the early stages of concept formation. Ronald Garcia was the supervisory author on this project and was involved throughout the project in concept formation and manuscript composition.

I was the lead investigator in the unpublished work reported in Chapter 6. Ronald Garcia was the supervisory author and was involved throughout the project in concept formation and manuscript edits. Reid Holmes and Margo Seltzer provided feedback as members of the supervisory committee during concept formation and study design.

All other chapters are an original, unpublished work by the author, Felipe Bañados Schwerter, as a lead investigator responsible for all major areas of concept formation, design, and manuscript formation. Ronald Garcia was the supervisory author on these projects and was involved throughout their development in concept formation and manuscript feedback and edits. All members of the committee provided feedback on the manuscript.

TABLE OF CONTENTS

Abstract . . . iii

Lay Summary . . . iv

Preface . . . v

Table of Contents . . . vi

List of Tables . . . viii

List of Figures . . . ix

Acknowledgements . . . xi

Dedication . . . xii

CHAPTER 1 Introduction . . . 1
- 1.1 Limitations of Abstracting Gradual Typing . . . 3
- 1.2 Thesis Statement . . . 6
- 1.3 Contributions . . . 8
- 1.4 Notes . . . 11

CHAPTER 2 Background . . . 12
- 2.1 Program Errors . . . 12
- 2.2 Tracking the source of a failure (or a computation) . . . 19
- 2.3 Gradual Typing . . . 32
- 2.4 Gradual languages by calculation: Abstracting Gradual Typing . . . 39
- 2.5 A brief history of error reporting in gradually typed languages . . . 41
- 2.6 Experimental Evaluation in Programming Languages . . . 46
- 2.7 Notes . . . 47

CHAPTER 3 From AGT to Precise AGT . . . 51
- 3.1 AGTLC: Applying AGT to the simply typed lambda calculus . . . 54
- 3.2 $\text{GTFL}_{\lesssim}$: A gradual language with records and subtyping . . . 74
- 3.3 Updating AGT: Evidence gets its own abstraction . . . 88
- 3.4 Precise Evidence for $\text{GTFL}_{\lesssim}$ . . . 90
- 3.5 Forward Completeness as a key to precision . . . 96
- 3.6 Notes . . . 98

CHAPTER 4 The essence of Galois slicing . . . 100
- 4.1 Example 1: a calculator (with randomness) . . . 102
- 4.2 A high-level view of a slicing system . . . 102
- 4.3 Applying the framework to the CARL language . . . 110

4.4 Example 2: adding recursive functions and substitution 115
4.5 Example 3: adding mutable state 122
4.6 Example 4: SCARF with a reduction semantics 130
4.7 Example 5: adding continuations with let/cc 132
4.8 Conclusion 136
4.9 Notes 137

CHAPTER 5 AGT that explains its failures 138
5.1 Slicing as error reporting 138
5.2 Slicing AGT at a high level 141
5.3 Slicing Elaboration for AGTLC 142
5.4 Slicing Evidence Composition for AGTLC 151
5.5 Slicing for the runtime language ARL 156
5.6 Slicing for Bounded Rows and Records 160

CHAPTER 6 Debugging run-time type errors in practice 175
6.1 The Wizard-of-Oz User Interface 176
6.2 Research Questions 177
6.3 Experiment Design 177
6.4 Selection of programs for user study 180
6.5 Results 184
6.6 Threats to validity 191
6.7 Conclusion 192
6.8 Notes 193

CHAPTER 7 Conclusion 194

BIBLIOGRAPHY 197

APPENDIX A $\mathrm{GTFL}_{\lesssim}$ Operator Definitions 209

APPENDIX B AGTLC Err-approximates Natural 222

APPENDIX C Galois slicing for natural semantics and for small-step semantics . . 227

APPENDIX D Defining evidence composition inductively 239

APPENDIX E User study design 241

LIST OF TABLES

Table 2.1 An example erroneous snippet in different static and dynamic type systems: Although all are valid syntax in their language, only the JavaScript case is a valid program. 14
Table 2.2 A comparison of the process required to obtain and use provenance in databases (with semirings) and in functional programs (with slices) 25

LIST OF FIGURES

Figure 1.1 Two program examples, one (left) in JavaScript, a dynamically typed language, and the other (right) in TypeScript, a statically typed language. . . . 2
Figure 2.1 A simple example Racket program with first-order contracts 16
Figure 2.2 An example Racket program with higher-order contracts 18
Figure 3.1 The Simply Typed Lambda Calculus with annotations, in a style ready to apply the AGT recipe 56
Figure 3.2 Gradual Surface Language for AGTLC 63
Figure 3.3 Runtime type system for ARL 71
Figure 3.4 AGTLC: Elaboration 72
Figure 3.5 The static type system of the STFL$_{<:}$ language. 76
Figure 3.6 GTFL$_{\lesssim}$: Static Semantics, part 1/2 78
Figure 3.7 GTFL$_{\lesssim}$: Static Semantics, Part 2/2 79
Figure 3.8 GTFL$_{\lesssim}$: Runtime Language Static Semantics 81
Figure 3.9 GTFL$_{\lesssim}$: Runtime Language Dynamic Semantics 83
Figure 3.10 GTFL$_{\lesssim}$: Elaboration 86
Figure 3.11 Decompositions and BRR Concretization Function 94
Figure 3.12 BRR Abstraction Function 95
Figure 3.13 BRR$_{\lesssim}$'s definition of well-formed evidence 95
Figure 4.1 The CARL language 102
Figure 4.2 Galois slicing definitions for the CARL language. 106
Figure 4.3 Specification of forward slicing for the notions of reduction in the calculator language 111
Figure 4.4 Forward and backward slicing multi-step operations 114
Figure 4.5 Extensions to the language to include first class functions 116
Figure 4.6 Forward slicing definition for substitution 118
Figure 4.7 Substitution backward slicing for the pre-substitution term slice . . 119
Figure 4.8 Substitution backward slicing for the replacing term slice 120
Figure 4.9 Extensions to the language to include mutable state 123
Figure 4.10 Notions of forward slicing for mutable state operations 124
Figure 4.11 Notions of backward slicing for mutable state operations 126
Figure 4.12 Extensions to the language to include continuations via let/cc . . . 133
Figure 4.13 Slicing rules for throw 134
Figure 4.14 Slicing rules for let/cc 135
Figure 5.1 (Total) initial evidence operation for AGTLC. 144
Figure 5.2 Forward slicing for (total) initial evidence in AGTLC, part 1 148
Figure 5.3 Forward Slicing for (Total) Initial evidence in AGTLC, part 2 150
Figure 5.4 Forward slicing for Elaboration for AGTLC 152
Figure 5.5 (Total) Evidence Composition for AGTLC 153
Figure 5.6 Forward Slicing for Evidence Composition in AGTLC, part 1 . . . 154
Figure 5.7 Forward Slicing for Evidence Composition in AGTLC, part 2 . . . 155

Figure 5.8 Forward slicing for the notions of reduction in ARL. 157
Figure 5.9 Initial Evidence for Bounded Rows and Records (selection of relevant rules for Row and Record Types) 163
Figure 5.10 Gradual Meet with BRR, part 1 169
Figure 5.11 Gradual Meet with BRR, part 2 170
Figure 5.12 Forward slicing for evidence composition with BRR 172
Figure 6.1 Tutorial program, highlighting the lack of type safety in TypeScript 179
Figure 6.2 Text editor interface used in the main study. 179
Figure 6.3 Heat maps of participant highlights - Program 1 186
Figure 6.4 Heat maps of participant highlights - Program 2 186
Figure 6.5 Heat maps of participant highlights - Program 4 186
Figure 6.6 Heat maps of participant highlights - Program 3 187
Figure 6.7 Heat maps of participant highlights - Program 6 187
Figure 6.8 Heat maps of participant highlights - Program 5 188
Figure 6.9 Histogram of total toggles of the TypeSlicer tool per individual task 190
Figure A.1 $\text{GTFL}_{\lesssim}$: Consistent Subtype Extrema 210
Figure A.2 Gradual Meet 211
Figure A.3 $\text{GTFL}_{\lesssim}$: Definition of Initial Evidence 211
Figure A.4 Consistent Transitivity: Part 1 215
Figure A.5 Consistent Transitivity: Part 2 216
Figure A.6 Definition of Initial Evidence for Bounded Rows 217
Figure A.7 Gradual Meet for BRR 218
Figure A.8 Static Subtype Meet and Join 219
Figure A.9 Consistent Subtype Extrema with BRR, part 1 220
Figure A.10 Consistent Subtype Extrema with BRR, Part 2 221
Figure B.1 Natural Small-step Semantics 223
Figure B.2 Evidence that supports type consistency 223
Figure B.3 Transformation from AGTLC into a Natural Semantics. 224
Figure B.4 Simulation relation between ARL and Natural 224
Figure C.1 Big-step environment based semantics 228
Figure C.2 Big-step substitution-based semantics 231
Figure C.3 Small-step substitution-based semantics 233
Figure C.4 Big-step environment based semantics, store extensions 235
Figure C.5 Big-step substitution-based semantics, store extensions 237
Figure E.1 Tool tutorial, phase 1/5 243
Figure E.2 Tool tutorial, phase 2/5 243
Figure E.3 Tool tutorial, phase 3/5 244
Figure E.4 Tool tutorial, phase 4/5 245
Figure E.5 Tool tutorial, phase 5/5 246
Figure E.6 Task interface, before running a program (TypeSlicer available) . . 247
Figure E.7 Task interface, after running a program (TypeSlicer is on) 248
Figure E.8 Task interface, after running a program (TypeSlicer is off) 249
Figure E.9 Task interface, before running a program (TypeSlicer not available) 250
Figure E.10 Task interface, after running a program (TypeSlicer not available) . 251

ACKNOWLEDGEMENTS

To Ron, thanks for your mentorship. You have provided endless support and guidance throughout the years. I will always miss our whiteboarding sessions and the reading group discussions. Seems like I managed to write a thesis without forcing the reader to know what a semilattice is. Hope this is sufficient.

To the rest of my dissertation committee, William, Reid, and Margo, thanks for your kindness and support through this research project, and for always posing interesting questions that took this work where I would not have imagined. I would also like to thank my external examiner David for taking the time to read this dissertation, and for your questions and feedback.

To Éric Tanter, thanks for introducing me to the field of programming languages, for your constant support and friendship, and for introducing me to Ron.

To every academic that kept their door open for me to drop by and chat throughout my higher education, thanks for your time. Besides those already mentioned, I would like to thank (chronologically) Alejandro Hevia, Nancy Hitschfeld, Alexandre Bergel, Claudio Gutiérrez, Jana Dunfield, Norm Hutchinson, Alan Hu, Ivan Beschastnikh, Steve Wolfman, Alex Summers, and Jorge A. Pérez.

To every friend and splabmate I have met at the Software Practices Lab and the Programming Languages Reading Group, especially to Dr. Nico, Dr. Joey, Rui, Paulette, Jonathan, James, Adam, Eric, Braxton, Yanze, Khurram, Dr. Arthur, and Chris. Thanks for making graduate studies worth it, for enlightening me with your vast knowledge of multiple research areas, and for the memes.

To St. John's College at the University of British Columbia, thanks for introducing me to my wife, for housing and feeding me for a considerable part of my degree, for providing scholarly connections beyond Computer Science, and for the many friends I have met there over the years (including Yasmin, Edmundo, Fabrício, Kate, Gerson, and Shams).

To my family and friends from Chile, thanks for all your support throughout these crazy years. Special thanks to Gustavo, Jorge, Alfredo, Matías, and the many other Felipes.

First, to my wife Marianne, to whom I owe everything.
Then, to my grandfathers Teo and Samuel, I miss you both.

To summarize: We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth (1974)

For if I am mistaken, I exist.

— Augustine (0426b)

1 INTRODUCTION

Programming has many challenges. For many decades, we have been facing a *"software crisis"* reflecting a *"gap between ambitions and achievements"* (Naur and Randell, 1968). Software projects routinely go over budget, get abandoned, and produce bug-ridden software. Programming is much more difficult when programmers do not know what they want the computer to do exactly, down to the last detail.[1] Unfortunately, this uncertainty is quite common, thus programming is usually a difficult task.

Programming difficulties are in tension against each other. Easing one difficulty can make other dimensions of programming more challenging. This tension impacts language design. As difficulties in using a programming language impact how difficult programming is, language designs can make programming both easier in one dimension and harder in others. Different language designs address complexities in tension by prioritizing some language facets over others.

An example of language designs in tension are statically typed and dynamically typed languages. On the one hand, statically typed languages guarantee that some classes of bugs never happen at run time. But to achieve this goal, the language pushes programmers to commit early to design decisions by declaring types that restrict the values that can be bound to each variable, passed to each function argument, or returned from function calls. On the other hand, dynamically typed languages do not require programmers to explicitly commit to type restrictions, but this flexibility allows more run-time errors.

Figure 1.1 shows two programs that demonstrate this tension. Both programs implement a dashboard for a car, one dynamically typed (left) and the other statically typed (right). Both programs follow a similar structure, with a `dashboard` function that calls a separate `range` function. Although contrived, these examples suffice to demonstrate benefits and disadvantages between these two language designs that manifest pervasively in larger software projects.

The dynamically typed program on the left is slightly shorter to write, but does not stop programmers from calling the `range` function with an incorrect argument (like giving the `fuel` directly, or providing a record missing the required fields). Programs that use `range` incorrectly will still run until the point of incorrect use. Programs produce an error when they reach that point. When prototyping, this may be desirable, as it lets programmers evaluate any behaviour that precedes the erroneous call.

The statically typed program on the right guarantees that every use of the `range` function receives appropriate arguments, at the cost of extra commitments both to the type discipline and to the type interface.

By committing to the type discipline, programmers are not allowed to run a program if any usage not accepted by the type checker appears in it. This universal safety requirement

```
7  type Gallons = number
8  type Miles   = number
9  type Car     = {
10     fuel : Gallons,
11     mpg  : Miles
12 }
```

```
1  function range (car) {
2      return car.fuel * car.mpg
3  }
```
A

```
4  function dashboard (car) {
5      return "Range :" + range(car)
6  }
```
B

```
14 function range (car : Car) : Miles {
15     return car.fuel * car.mpg
16 }
```
C

```
17 function dashboard (car : Car) : string {
18     return "Range :" + range(car)
19 }
```
D

Figure 1.1: Two program examples, one (left) in JavaScript, a dynamically typed language, and the other (right) in TypeScript, a statically typed language.

is great for shipped code, but the requirement may be too restrictive when debugging or testing. Programmers may want to test particular contexts where they are certain the unaccepted uses will not be reached. Programmers may also want to check what a program does up to the point of unaccepted use, to think of alternatives. The statically typed language puts the type discipline first and does not let programmers do any of this: before running the program, every type rejection must be fixed.

By committing to the type interface, programmers expose design decisions that constrain future changes to the program. In our example, the code exposes what a car is, and future changes to the interface (say, using `Litres` instead of `Gallons`, or accommodating batteries in hybrid or electric cars) may require major global changes to the program, as changing the interface at one point of the program (say, if `range` becomes an object method instead of a function) may stop the program from compiling until all changes are made, including `dashboard` and any other external uses of `range`.

The tension between these two designs is not unique to programming languages, but applies more broadly to all interfaces. Human-computer interaction researchers study interfaces and their tensions. Their insights also apply to language design, as programming languages are an interface between programmer and programmed. T. R. G. Green (1990) identified a general tension over the *viscosity* dimension in interface design, where the designer must choose how much work is needed when an interface user changes their mind: *"... one can get the program (or whatever) set up fast,* or *one can be able to change it quickly, but getting the best of both worlds requires really clever design."*

One instance of such clever design is gradual typing. In gradually typed languages, programmers can combine both statically typed and dynamically typed programming styles in the same program. By using gradually typed languages, programmers can control the migration between run time and compile time checking of properties without switching languages. For example, in a gradually typed (or just "gradual") language programmers can combine parts of the left and right programs from Figure 1.1. A programmer can write the dynamically typed version of `dashboard` (box B), and combine it with the statically typed version of `range` (box C). If the programmer then wants to change the `Car` interface while prototyping, she does not need to touch the `dashboard`

function at all: she can choose how far to commit to the type discipline and interfaces as she designs them.

Gradually typed languages were originally intended as a technique where dynamically and statically typed code can be written and interact in the same program. Combining both kinds of code allows us to reap the benefits of both associated programming disciplines: rapid prototyping (the dynamic advantage) and declaring type invariants that can be guaranteed and relied upon (the static advantage) can happen together.

The use of gradually typed languages has expanded beyond the combination of static and dynamic, as designers recognized this was not the core contribution of gradual typing. The core contribution of gradual typing is to let language designers choose the degree of flexibility they want to give programmers over the checking of the invariants in their programs. Combining static and dynamic gives programmers full flexibility, but the language designer may also decide that some invariant checks should never be postponed. The programmer may decide whether to enforce *some* invariants at run time or at compile time, but only after the language designer decided *which* invariant checks can be delayed.

The language designer can choose not only which preexisting static checks to postpone, but can also choose to introduce new static checks that a language did not previously enforce. Gradual typing allows for a programmer-controlled migration to new language features that affect the type discipline of the language (usually by making it stricter) or introduce new kinds of type interfaces. A recent example in this dimension is the introduction of gradual dependent types, where a statically typed language can be extended with dependent types in a way that allows for incremental commitment to the dependent type discipline, as described by Eremondi (2023).

Language designers have applied gradual typing ideas to increasingly advanced language features. Advanced features force gradual language designs to get cleverer and harder to get right. This increasing complexity showcased the need for design guidelines for gradual languages. One set of guidelines, the Abstracting Gradual Typing (or AGT) framework introduced by Garcia, Clark, and Tanter (2016), provides a recipe for designing new gradual languages. This recipe lets designers take the specification of a statically typed language and produce a gradual version by focusing only on a few relevant design choices: the designer only has to choose a syntax for gradual types, the meaning of their choice of gradual types and how these connect to static types. Without the recipe, language designers must also specify and justify gradual versions of the static checks performed in the original language, the run-time representation of gradual information, and run-time checks that propagate this information as programs run. The recipe reduces the work for a designer to provide a gradual language specification by defining these elements in terms of the recipe's inputs.

1.1 LIMITATIONS OF ABSTRACTING GRADUAL TYPING

By simplifying the design process, AGT has encouraged considerable research on the design of increasingly complex gradual languages (Bader, Aldrich, and Tanter, 2018; Eremondi, Tanter, and Garcia, 2019; Labrada, Toro, and Tanter, 2022; Lehmann and Tanter, 2017; Lennon-Bertrand et al., 2022; Toro, Garcia, and Tanter, 2018; Toro, Labrada, and Tanter, 2019; Toro and Tanter, 2017, 2020; Vazou, Tanter, and Van Horn, 2018; Wise

et al., 2020). Although the AGT recipe provides many benefits to language designers and users, the recipe also has limitations. This dissertation addresses two limitations of the AGT framework. First, AGT can generate a gradual language where some type information is lost at run time. Some programs which should produce a run-time error instead run successfully to a value, because the runtime misses inconsistencies between type constraints. To avoid this issue, I restrict the AGT recipe's inputs to *forward-complete* operations (Giacobazzi and Quintarelli, 2001). Second, programmers using AGT languages get no feedback when a run-time check fails in their programs. I extend the AGT recipe, providing program slicing systems to debug run-time check failures. I now discuss these limitations, starting by the second one.

1.1.1 *AGT and run-time type error reporting*

One of the limitations of AGT that I address concerns run-time type error reporting. Some programming mistakes are not immediately visible to the gradual type system. The gradual type system cannot identify some type inconsistencies immediately, because it is designed to accept programs whenever there is a chance for type interfaces to be consistent, instead of the static typing approach of rejecting programs whenever there is a chance of inconsistency. As the program runs, the runtime gains more precise information that can reveal an inconsistency that was invisible when the program was type checked.

Let's see an example of a type inconsistency identified at run time. Imagine a programmer writes the car dashboard program by combining the dynamically typed `range` (box Ⓐ) from Figure 1.1 with the statically typed `dashboard` (box Ⓓ). Later, the programmer needs to change the `Car` interface and its uses to have the following type:

```
7  type Litres = number
8  type Kilometres  = number
9  type Car = {
10   fuel : Litres,
11   kpl  : Kilometres
12 }
```

The programmer also forgets to update the code in `range`. The program will still compile and run, because box Ⓐ does not place constraints on what `range` expects and produces. However, any time the `dashboard` function runs it will produce an error, as the field `mpg` does not exist anymore and cannot be accessed in line 2.

A gradual language designed using the AGT recipe detects this inconsistency and aborts the program. Gradual languages accept programs when the type checker cannot immediately see a type inconsistency, but if the gradual language is well designed, it "trusts, but verifies", introducing run-time type checks wherever needed to guarantee that inconsistencies will be revealed as run-time type errors instead of leading to behaviour inconsistent with the limited type discipline the programmer committed their programs to follow.

Unfortunately, on detecting an inconsistency, the AGT-based language will abort without any specific feedback beyond a generic "Error" message, and this is a problem.[2] While a recipe for gradual languages does not have to provide detailed feedback for run-time type errors *in theory* (to be type safe), it would be nice to go beyond a generic "Error" message *in practice* for programmers to have a more useful language.

A run-time type error is the reflection of a programming mistake.[3] Types are part of the program's code, so they can also have bugs. For programmers to be able to rectify their mistakes, they must first understand them, and an abstract "Error" does not help them. To understand run-time type errors, it will help programmers to have access to information relevant to the type inconsistency, so they can alter the program in whichever way they see fit.

One kind of relevant information is *provenance*,[4] a representation of the origin and custody of objects. Provenance can be used to identify the parts of a program involved in the inconsistencies that the language detects in a run-time type error, without classifying individual constraints as right or wrong, or making any assumptions about the correctness of programmer-introduced type constraints.

Although provenance information is usually represented as a graph, it can also be represented in a program as a *slice*, where parts of the program are marked as not related to the failure itself. For example, from the program trace that runs our incorrect program to an error, provenance can be used to compute the following slice of the original program, highlighting that type `Car` does not have an `mpg` field.

```
1  type Car = {
2    fuel : Litres,
3    kpl  : Kilometres
4  }
5  function range (car) {
6    [redacted] car.mpg
7  }
8  function dashboard (car : Car) [redacted] {
9      [redacted] range(car)
10 [redacted]
```

Although in this example it looks like the problem would be fixed by changing `car.mpg` to `car.kpl`, this is a decision the programmer needs to make, not the compiler: maybe the type of `Car` should keep an `mpg` field, or maybe the parameter given to `range` should be transformed before calling the function, the type system cannot know. There is no "right change" in the code itself, only an inconsistency. The programmer is responsible for deciding how to change the program and remove the inconsistency using external information.[5] The language design itself cannot blame or redeem a type definition (which could itself be wrong), but it can ensure the programmer has access to all the type assumptions leading to the inconsistencies that produced a particular run-time failure: it is not the language who judges a programming mistake, but the programmer.

1.1.2 *AGT and the precision of run-time type information*

It is not just programmers who can make mistakes, but language designers also make mistakes. The original AGT framework does not mandate precise run-time abstractions. But imprecise abstractions impact the soundness of the gradual language with respect to the original statically typed language used in the recipe. We expect a recipe for gradual languages to produce languages that always respect the type constraints of the original static language, yet this is not always guaranteed in the original AGT framework.

Let's see an example of this subtle limitation. Imagine a different programmer who wrote the car dashboard by combining the dynamically typed version of the `range` function from Figure 1.1 (box Ⓐ) with the statically typed `dashboard` function (box Ⓓ), and then removed the `mpg` field from the type of `Car`. That is, they removed line 3 in the following program:

```
1  type Car = {
2    fuel : Litres,
3    ~~mpg : Miles~~
4  }
5  function range (car) {
6    return car.fuel * car.mpg
7  }
8  function dashboard (car : Car) : string {
9    return "Range :" + range(car)
10 }
11 dashboard ( { fuel : 25, mpg : 30 } )
```

Because the programmer only changed the `Car` interface, the `Car` objects passed around still contain the field `mpg`, like the argument to `dashboard` in line 11. However, the field `mpg` should be now inaccessible when the values are seen as a `Car`. One would then expect this change to force the program to always produce a run-time error, as even though `range` is dynamically typed, it must respect the type constraints coming from its environment (in this case, from line 18 that forces the argument to be a `Car`, which now *only* has `fuel` as a field).

The original AGT recipe does not always catch these kinds of mistakes. In the original language with records proposed by Garcia, Clark, and Tanter (2016), this program would not produce a run-time error. If a value passed as an argument to `dashboard` contains the field `mpg`, the semantics of gradual types they propose *forgets* part of the type constraints and will let the program run to completion. The problem is that the original recipe does not sufficiently specify the requirements for run-time information propagation, so this kind of abstraction leakage could remain undetected throughout evaluation.

1.2 THESIS STATEMENT

This dissertation addresses these two limitations. My thesis is that **When improved with program slicing and forward-completeness restrictions, the Abstracting Gradual Typing recipe can address some of the mistakes made by programmers and by language designers. Forward-completeness ensures that type inconsistencies produce run-time type errors, and provenance-based program slicing ensures that those run-time type errors can be traced back to the program fragments that might have contributed to the inconsistencies.**

Some of the type constraints in a gradual program may not be immediately visible to the type checker. Thus a program might run to a point where type constraints that cannot hold together are required to hold together. At that point, the language must detect the inconsistency. I believe it would benefit programmers if the programming system identifies the parts of the program that might have contributed to the problem, and rule out the parts that can be proved to not be involved.

A gradual programming language intends to provide a *"programmer-controlled migration"* (Siek and Taha, 2006) towards a type discipline by choosing how far to commit to it. I argue that in a programmer-controlled migration, only the programmer *controlling the migration* can tell which constraints are right or wrong, and that language designs that let inconsistencies sneak through evaluation should not be allowed by the AGT recipe.

Before discussing the contributions of this dissertation, I spend the rest of this section discussing some of the related ideas connected to my work.

1.2.1 *Run-time type error feedback in gradual languages*

One key property of sound gradual languages is that interface assumptions made by the static language should be permanent. This affects program evaluation, which must check whether any incompatible assumptions are made. To achieve this goal, the semantics includes checks that may stop execution. The language runtime stops execution when it discovers a place in the program that relies on contradicting static assumptions: it cannot continue because the type interface is impossible to fulfill. For example, one module may expect a fuel-based `Car` object while another module may expect a battery-based `Car` object, with incompatible interfaces. A dynamically typed object that is passed through these modules commits to both interfaces but cannot fulfill both at the same time.

To comply with this design requirement, it suffices to stop execution with an opaque "Error" output message with no details when an inconsistency is found.[6] That is *formally* sufficient, but if a programmer wants to change the program into another where the check does not fail (a "fixed" program), they must debug the type inconsistencies. It will be harder for a programmer to debug a program for inconsistencies without any information about what the inconsistencies are.

Current gradual languages give run-time error feedback by subscribing to a *blame semantics*. In a blame semantics, all run-time checks in the program are annotated with labels. These labels denote places in the original program. The run-time semantics propagates these labels while checking invariants as execution proceeds. If an invariant check fails, the system aborts execution and reports a program location said to be "blamed" for the failure.

Can programmers trust that a blamed location is the source of the failure, a place in the program that needs changing? Unfortunately, not always. The current guarantee that designers establish about all error reports in a blame semantics is the *blame theorem* as introduced by Wadler and Findler (2009), but this theorem does not directly guarantee properties about failures. The blame theorem specifies only the subset of all run-time checks that will never produce an error and does not directly evaluate the relevance of a reported location with respect to any type failure. I discuss the details of the blame theorem in Section 2.5.1 and further limitations of blame in Section 2.5.3.

Although the blame theorem acts as a design guideline for blame semantics, there are still many approaches to propagate and manage these error labels: at least seven different ways[7] to decide whether evaluation may continue and how. The majority of these have an associated blame semantics and a blame theorem.

Semantics with blame have generally proven that "well-typed programs can't be blamed" (Wadler and Findler, 2009), which implicitly assumes that types have no bugs. From this assumption, designs in the literature claim that it suffices to point to a single place in a program. In some cases, unfortunately, the locations reported by blame are not immediately relevant to a failure. Recent guidelines for debugging refer to "blame trails" (Lazarek, Greenman, et al., 2021; Lazarek, King, et al., 2019), an iterative process of refining the error message by changing the program, forcing the developer to further commit to deeper type constraints. While the process is intended to eventually force the semantics to provide a relevant failing point, that is not always guaranteed to happen.

Moreover, the assumption that all type declarations provided by the programmer have no bugs does not hold in practice: type annotations are also code, and just like any other piece of code, they are regularly wrong.[8] The current solution seems to be to remind developers of the correctness assumption in the error message.[9]

A separate design guideline for blame semantics has been Complete Monitoring (Dimoulas, Tobin-Hochstadt, and Felleisen, 2012; Greenman, Felleisen, and Dimoulas, 2019), a property that guarantees that a run-time semantics inserts and preserves sufficient checks. Complete monitoring only justifies "when" a failure needs to happen, not the error message nor the reported program locations, which are usually justified by other means.[10] Although complete monitoring was originally intended to subsume correct blame, I discuss how complete monitoring can be discussed separately from blame in Section 2.2.4.

1.2.2 *Provenance*

This dissertation shows that provenance can be applied to overcome limitations in error reporting for run-time type checking in gradual languages.

Provenance is meta data that a system must collect to be able to answer questions regarding an artifact's history (Carata et al., 2014). Here an "artifact" may range from the output of a query on a database to the result of running a program. Formal theories of provenance arose in the database community, but since then have been investigated in the programming languages community as well, beginning with work by Cheney and others to recover the slice of a program that was involved in generating a result (Perera, Acar, et al., 2012; Ricciotti et al., 2017).

1.3 CONTRIBUTIONS

I take ideas from provenance to develop a formal framework for tracking and querying the provenance of run-time type information, in particular run-time type errors. I introduce a recipe to apply this provenance technique to a gradually typed language semantics designed using AGT. The recipe mechanically produces a slicing semantics for the gradual version of the language.

A slicing semantics formalizes the concept of run-time type provenance by generating and connecting program slices over a program trace to the original program. A program slice (see Section 2.2.1) is a portion of a program deemed to have been involved in a computation, be it successful or a run-time failure. The formalized semantics accepts a set

of provenance queries that can be performed *post-hoc* on the trace of a failing execution, with the goal to aid debugging of run-time type errors.

I base this work on the AGT framework of Garcia, Clark, and Tanter (2016) to make this provenance model widely applicable, both to the many kinds of semantics that have been designed using AGT and to those that will use AGT in the future. In other words, I extend the AGT recipe to account for error reports. I apply this general model to a particular programming language, BRR (for Bounded Rows and Records), and provide a prototype implementation of its language semantics. AGT is a good starting point for our model, because it explicitly links gradual types and their run-time behaviour to a statically typed specification. This provides for a more direct approach than blame and complete monitoring, both of which rely upon an interpretation of the labelling discipline in a language. Although the labelling discipline aims to connect to the gradual types in the original program, it does so only indirectly.

The development of BRR also identified relevant limitations in the way that the original Abstracting Gradual Typing recipe handles imprecise run-time type information abstractions. In summary, when using the unrestricted original system, some run-time type information abstractions may be too coarse, and relevant type information may get lost as evaluation progresses. Losing information allows some programs to break the type invariants assumed by the original program without detection, and evaluation can continue without reporting the inconsistency to the programmer. To solve the issue, I revise the framework to require that all operations dealing with run-time type information follow specific precision constraints. These "forward-completeness" constraints guarantee that propagating run-time information does not reintroduce imprecision.

I then add provenance to this updated model of AGT. With provenance, one gets stronger guarantees about what is involved in a failure: once the runtime detects a failure and I obtain a program slice via provenance, it is guaranteed that any change outside the slice does not impact this particular inconsistency. No type conversions are deemed safe *a priori* (as a blame theorem would characterize), even though the semantics of provenance and gradual typing induces a notion of safe conversions in particular execution traces.

Although the provenance semantics does not directly require forward-completeness, the extra constraints guarantee programs with inconsistencies will produce errors. One can then use provenance to obtain a slice from the program trace that leads to the error.

In addition to the theoretical development of the provenance framework, I perform a user study to evaluate the applicability of these extensions to type debugging. Thus this research program intersects between programming language research, type systems, and software engineering. The contributions in this thesis are:

- **An improved understanding of limitations in the design of AGT languages and a refinement of the requirements to guarantee precision and reduced space usage when using AGT**. I developed a new semantics for the concrete language used as an example in the original AGT paper by Garcia, Clark, and Tanter (2016): A gradual language with records and subtyping. This process required the introduction of new gradual types for records and the identification of limitations in the original formulation of AGT. I describe stronger constraints to the framework such that languages guarantee precise enforcement of declared type invariants. This work has been published in Bañados Schwerter, Clark, et al. (2021).

- **An adaptation of run-time provenance to small-step semantics**, adapting the notion of program slicing by Perera, Acar, et al. (2012) and Ricciotti et al. (2017) to account for arbitrary small-step semantics. I use this transformation to introduce the technique of program slicing by example, using the framework to provide run-time slicing to a tower of increasingly complex interpreters, from a calculator up to a functional language with first-class continuations. This progression of languages demonstrates how the approach scales to many language semantics.

- **A formal model for run-time gradual type provenance**, extending the notion of provenance previously developed to account for gradual type information and to specify how to ask type provenance questions in the context of run-time errors. Given an AGT-based semantics for a gradual language, one can systematically extend it with support for provenance tracking and querying. This model includes a formal semantics for run-time gradual type provenance, which collects sufficient run-time information to answer the provenance questions specified in the model. I develop a formal system that conservatively extends the AGT framework to collect run-time type information traces. This provenance system extends the run-time semantics to accept program portions (slices) in a *forward slicing* semantics that is consistent with the original run-time semantics.

  From a trace of a program that ends in a run-time type error, slicing computes the portions of the original program generating the run-time type information relevant to the failure. This program slice still reproduces the failure, while providing a smaller surface for debugging than the complete program. The slice provides more feedback than a single-location blame report, sufficient information to reproduce the failure, and is a novel application of provenance to the domain of gradually typed languages.

- **A proof of the guarantees provided by the provenance trace, its properties and its error reports in terms of the model for gradual type provenance**. I extend the theorems of consistency and minimality introduced by Perera, Acar, et al. (2012) to apply to the run-time semantics of gradually typed languages and their error reporting. Also, inspired by the gradual guarantee theorems of Siek, Vitousek, et al. (2015), I show that any changes to programs performed on the non-preserved portion of the slice cannot stop failure: the inconsistencies that produce the failure are contained in the program slice recovered by slicing.

- **An exploratory analysis of how developers debug types with program slices in gradual languages, by way of a user study based on a prototype slicing tool**.

  I performed an exploratory qualitative user study using the slices provided by BRR and a prototype slicing tool. The study provides evidence that slices can be a useful tool for debugging run-time type errors in gradually typed languages.

  The prototype tool presents program slices as an overlay over a program's source code in a TypeScript-like language (modified to have a sound gradual semantics). I chose TypeScript syntax as a candidate of interest because of its high popularity,[11] which was expected to facilitate developer recruitment for the user study.

1.4 NOTES

1. As noted by Brooks (1975), *"The hardest single part of building a software system is deciding precisely what to build."*
2. For readers familiar with gradual typing literature, I discuss connections with *blame semantics* in Section 1.2.1. In short, this dissertation proposes using program slicing systems as an alternative to blame semantics.
3. Outside the scope of pedagogical examples intended to show incorrect behaviour.
4. The idea of *provenance* originates on the information that is used to authenticate works of art.
5. Considerable research focuses on making external information available to programmers, including Viviani (2022) and Marques (2022).
6. Though one may view this requirement to stop execution as modest and reasonable, many languages do not do so. They have good reason though: the obvious means to satisfy this requirement imposes prohibitive overhead, as discussed in Section 2.3.4, and some languages consider the extra guarantees to not outweigh the added cost in enforcing them.
7. Greenman, Dimoulas, and Felleisen (2023) comparatively describe six semantics, "Natural", "Co-Natural", "Forgetful", "Transient", "Amnesic", and "Erasure". The seventh in the group is the semantics provided by Abstracting Gradual Typing, "Supernatural".
8. According to recent research, manually associating type annotations is considerably error-prone (51% accurate, $\pm$ 8.5%), see the work by J.-P. Ore, Detweiler, and Elbaum (2021)
9. In Typed Racket, the blame message "warns the developer on the last line with "assuming the contract is correct."" (Greenman, Felleisen, and Dimoulas, 2019). This behaviour is inherited from the Racket contracts used to implement run-time gradual type monitoring in Typed Racket. The authors recognize this behaviour as useful for debugging via examples while recognizing it is unusual, as others gradual languages rely on the type correctness assumption.
10. Proving complete monitoring usually relies on propagating labels. All current uses of complete monitoring are refinements or justifications of blame semantics, so they reuse the labelling already available for blame propagation.
11. TypeScript appears as the 5th most popular programming language in the 2023 Stack Overflow Survey (`https://survey.stackoverflow.co/2023/#section-most-popular-technologies-programming-scripting-and-markup-languages`)

~~*For if I am mistaken, I exist*~~ *For if I am* deceived, *I am.*

— Augustine (0426a)

2 BACKGROUND

This chapter contains the background material that a non-expert reader needs to know to understand the chapters that follow. Because I provide my particular point of view on these topics, I hope that this chapter will prove useful for experts as well.

I structured this chapter into six sections. Section 2.1 discusses **program errors** and two approaches to address the occurrence of some errors in programs: types and behavioural contracts.

Section 2.2 focuses on techniques to **identify where failures come from**: program slicing, provenance and contract blame. Each of these techniques aims to ascribe a failing result to parts of the original program, which is my goal in error reporting.

The next three sections discuss different dimensions of **gradual typing**, the space of programming language design I have set for my research. Section 2.3 discusses both the evolution of gradual typing and some key questions that led to the development of the AGT framework: *What is "gradual" typing, after all?*, and *How do we make languages "gradual" in general?*. Section 2.4 presents the AGT framework. Section 2.5 connects the previous sections by diving into error reporting for gradual languages.

Section 2.6 situates our user study in the context of broader **experimental evaluation in programming languages**, bringing up related studies of gradual typing and error reporting.

2.1 PROGRAM ERRORS

Sometimes computer programs produce results unexpected by the programmer. For example, a program test can fail due to an implementation bug. A program can also fail when a user provides inputs the programmer considers invalid or unexpected. The programmer decides what the program does in those cases, usually by aborting execution with a program error (including some error message). Sometimes computer programs also produce program errors unexpected by the programmer.

In particular, programming systems[1] sometimes produce unexpected results. Programming system designers choose what the system does when a programmer writes a program inconsistent with the language design. In so doing, they define what will be a programming error in the language.

The quality of the feedback that programming systems provide in their error messages is not uniform. One implementation may just abort execution with a generic "failure" message, another may provide detailed feedback including the place an error happened, on the hope the information would be of use to the programmer. These differences arise

even when programming systems have much in common, say two implementations of the same programming language.

Error messages can also be misleading to programmers. Programmers' intentions are many times ambiguous and change. Their intentions are not based only on the information present in the failing program. Programmers also rely upon external information they have access to, including undocumented information they know. The language designer cannot read the minds of programmers to extract all the information needed to see what the programmer truly meant, thus systems can detect errors only up to a certain point:

"A computer system is not aware of human intentions. There will always be human mistakes that the system cannot recognize as errors. Despite this, there are many that it can *recognize, and its design will determine* which *human mistakes can become detectable program errors" (Jakubovic, Edwards, and Petricek, 2023)*

Error messages can be precise only up to a point. Making error messages too specific, for example by proposing specific changes to a program, risks disagreeing with the intentions of the programmer (Becker et al., 2019),[2] thus may make error messages misleading, directing developers to introduce changes where they shouldn't (Barik et al., 2017).

Error messages can also be misunderstood by programmers. For example, students *"often do not understand the vocabulary of messages, and anyway, obey an "edit here" mentality based on what code is highlighted"* (Wrenn and Krishnamurthi, 2017).

Errors may sometimes not be detected immediately, and an error message may point to a place in the program far away from the true source of a problem, showing instead where symptoms are manifested (Becker et al., 2019). This distance can also make error messages misleading or misunderstood.

Despite their limitations, error detection tools are useful and applicable at different stages of the development cycle.[3] Programmers do spend a considerable portion of a debugging task (13-25%) reading error messages (Barik et al., 2017), a task that appears to be harder than reading source code.

Some language features are intended to help programmers identify their programming errors. Their implementation checks that the programmer is consistent and that the program is doing what the programmer wrote. I discuss two approaches used in programming languages to impede erroneous behaviour: types in Section 2.1.1 and contracts in Section 2.1.2.

2.1.1 *Types*

Types are one approach for guaranteeing that certain classes of errors never go undetected. A *type* is an abstract representation of a property that holds for some syntactically valid expressions in a programming language. A *type system* or *typing judgment* is a formally specified logical judgment that associates a syntactically valid expression in a language with a particular type, encoding invariants that must hold about that expression. In practice, most people do not interact with a type system directly but through a type checker. A *type checker* is an implementation of a type system that determines whether a program satisfies the typing judgment.

```
#include <stdbool.h>

bool f (bool x){
  if (x){
    return x;
  } else {
    return x();
  }
}
```

C99

```
let f x =
  if x then
    x
  else
    x();;
```

OCaml

```
let f =
function(x){
  if (x) {
    return x
  } else {
    return x()
  }
}

module.exports =
  { f : f }
```

JavaScript

Table 2.1: An example erroneous snippet in different static and dynamic type systems: Although all are valid syntax in their language, only the JavaScript case is a valid program.

Programming language theory uses a specific syntax to describe logical judgments, based on the work of Hertz (1922, 2012) via Gentzen (1932, 1969). We use this syntax throughout the thesis to describe type systems and other logical judgments. The syntax represents if-then style statements, specifically "if $premise_1$, ... , and $premise_n$, then *conclusion*", as an *inference rule*:

$$\text{[Rule-Name]}\frac{premise_1 \quad \ldots \quad premise_n}{conclusion}$$

This notation represents a constructor for proofs of the judgment at hand. To build a proof of the conclusion one must provide, as arguments, proofs of all the premises. It is common practice to also provide a name for these rule constructors, next to the horizontal line. These constructed proof objects are called *derivation trees*.

2.1.1.1 *Types and programming languages*

Programming languages can be classified according to their use of types. If the runtime semantics of a language is defined only for programs that have already been accepted by the type checker, the language is *statically typed*[4] (Cardelli, 1997). The type checker for a statically typed language rejects those candidate programs it cannot guarantee will preserve the specified invariants, stopping the compiler without producing an executable. These rejected programs are not given any semantics. Statically typed languages include those where the programmer must write explicit type annotations (like in Java and C) and languages where the type information may be inferred, so programmers do not usually need to write annotations (like in OCaml and Haskell). When a language allows for the execution of programs without checking invariants beforehand, the language is *dynamically typed*.[5] In a dynamically typed language, if any type invariants exist, they must be checked during execution. What type invariants are actually enforced varies from language to language.

The main difference between static and dynamically typed languages is when invariants are checked, not whether a program includes type annotations. This distinction can be exemplified with the programs in Table 2.1, which describe an equivalent code snippet in

three languages: C99 (statically typed with annotations), OCaml (statically typed with inference), and JavaScript (dynamically typed). The first two (statically typed) reject these programs before running them, because the type system can identify an inconsistency between the uses of `x` in the `if` clause. The variable `x` must be at the same time a Boolean and a function, which are disjoint categories: the variable must be a Boolean either because of an annotation (C99) or because it is used as the test for an `if` expression (OCaml), and it must also be a function because it is called as such in the `else` branch. Both the C and OCaml type systems reject the program. Contrast this behaviour with JavaScript, where the snippet is accepted and compiled, but produces a run-time type error if a program calls `f(false)`.

Neither statically typed languages nor dynamically typed languages have a clear advantage for every possible context, at least in the anecdotal experimental evidence available. Mayer et al. (2012) performed a controlled experiment where 27 people completed some tasks faster using a static type system, while for other tasks, the opposite held. Kleinschmager et al. (2012) argue that static type systems improve the maintainability of software systems, *except* for fixing semantic errors, where they *"observed no differences with respect to human development times"* between both language kinds in a 36-participant controlled experiment, a sample formed mostly by students.

A basic property that a statically typed language should have is *type safety*. This theorem is usually connected to the 45-year-old motto coined by Milner (1978), that *"well-typed programs cannot go wrong"*. Milner's motto is misleading, as well-typed programs *do* sometimes go wrong: for example, they can produce a run-time error. The fine print in the motto is that Milner had a quite specific meaning for "wrong", a special catch-all state in his language. He defined the semantics of his language in such a way that any program containing the bugs he wanted to avoid would run to the "wrong" state. These buggy programs would be rejected by the type checker, thus the motto. But as discussed in Section 2.1, a type system is limited in its ability to distinguish right and wrong. A safe type system provides a weaker guarantee than "never going wrong", it guarantees that evaluation is well-defined for all well-typed programs. Wright and Felleisen (1994) framed this guarantee in a more precise motto, that well-typed programs *"cannot get stuck"*.[6]

Although dynamically typed languages may also have a semantics where programs do not get stuck,[7] they allow for more program bugs to remain in software, hiding until a particular run. Z. Chen et al. (2020) collect examples of risky dynamic typing behaviour, which is likely to introduce bugs. These examples include *"inconsistent typing practices"*, where arguments and variables are used as (or assigned to) types that are not structurally equivalent nor related by structural subtyping. Programmers have to introduce *ad hoc* type checks in programs to ensure values have the needed types. Gao, Bird, and Barr (2017) evaluate whether adding type annotations would have helped to identify bugs in dynamically typed code: they take a set of (solved) bug reports in some popular JavaScript repositories and check whether having added type annotations before would have identified the failure,[8] finding that adding type annotations can prevent 15% of the bugs in some public projects on GitHub.[9]

Reaping the benefits of both static and dynamic typing in a single language is what motivated the development of gradual typing, which I discuss in Section 2.3.

```
1  (define/contract abs
2    (-> number? (>=/c 0))
3    (λ (n)
4    (if (≤ 0 n)
5      n
6      (- n))))
7
8  (define/contract wrong-abs
9    (-> number? (>=/c 0))
10   (λ (n)
11     n))
12
13 (abs -10)  ;; This is ok
14 (wrong-abs 42) ;; This is ok
15 (wrong-abs -10) ;; This must fail
```

Figure 2.1: A simple example Racket program with first-order contracts

2.1.2 *Higher-order behavioural contracts*

Another approach to detect certain program errors is to use behavioural contracts. Although our work on gradual typing depends on types, which do not always need the expressivity of contracts, major gradual typing implementations like Typed Racket (Tobin-Hochstadt and Felleisen, 2008) rely upon contracts as a run-time checking tool, and the theoretical support for error reporting in behavioural contracts has been the basis for prior error reporting mechanisms for gradually typed languages. We therefore need to discuss them.

Contracts are a language feature originally proposed by Meyer (1992)[10] to declare assertions about program components like procedures. Assertions are predicates that represent a specification, written in the same language as the rest of the program. These assertions may represent pre-conditions, post-conditions, or invariants. I focus on the first two. Pre-conditions are predicates that must hold before the component is executed, thus the component can rely on them. Post-conditions are predicates that must hold after the component is executed, thus the component must behave consistently with them.

Beugnard et al. (1999) named these pre- and post-condition checks as "Behavioural contracts", to distinguish these contracts from nonnegotiable contracts over components, like the language syntax, and from more negotiable contracts that cannot be defined as pre- and post-conditions, like synchronization or quality-of-service contracts.

These assertions cannot always be immediately resolved where they are declared. When we consider functions as components, pre-conditions and post-conditions may rely on the particular arguments given to a function call, therefore they cannot be checked until we call the function and need to be re-checked on each separate function call.

Figure 2.1 shows an example of a program using first-order contracts in Racket, similar to the language in Findler and Felleisen (2002). Although the contracts need to be checked on every use of the function, they can each be immediately checked, because the inputs are either a `number?` or they are not, and the outputs are either positive or not, as `>=/c` is a higher-order function that takes a number as an argument and produces a new contract

function checking whether a value is greater or equal than the given argument, in this case, zero. The function `wrong-abs` is sometimes inconsistent with its contract, as it is just the identity. When we call the function with a positive number, the contract check succeeds. When we call the function with a negative number, the contract check can immediately fail.

When the runtime detects that an assumed contract does not hold, it aborts computation: if a pre-condition fails, the function should not be executed as its assumptions may not hold. This implies a bug in the context that uses the component. If a post-condition fails, the function does not fulfill its own specification. This implies a bug in the function itself.

When execution aborts due to a broken contract, we would like to identify the broken contract for debugging purposes. The simplest way to identify a broken contract is to mark every contract in the program with a unique label (for example, the line on which the contract was declared), and report one of these labels as the culprit when we abort computation.

How does the run-time mechanism decide which contract is broken? If the language is strict and only permits contracts that can be immediately checked, then it is easy to just point at the line where the failing assert was checked. If the assert is a method pre-condition, then the error should be traced to the *call site* of the method, as the arguments given did not comply with the specification. If the assert represents a post-condition, then the error should be traced to the *body of the method*, as even though our arguments fulfilled the precondition specification, running the method did not comply with its promised invariant.

Sometimes programmers want to write contracts that cannot be immediately checked: for example, when language values include functions. If a function can return a new function or receive a function as an argument, you might in general want to express pre-conditions and post-conditions about those function values, their inputs and outputs. These contracts are called *higher-order behavioural contracts*.

Higher-order contracts, introduced by Findler and Felleisen (Findler and Felleisen, 2001, 2002; Findler, Latendresse, and Felleisen, 2001), can enforce contracts for function values and assert properties required of arguments when the function is called, like "the first argument is a non empty list", as well as contracts about the result of calling the function, like "the function always returns positive values". The main difference with first-order contracts is that these cannot be immediately checked: the contracts must wait to be checked until the function values are called.

I introduce an example program making several uses of these higher-order contracts in Figure 2.2. The example also uses Racket, with a syntax similar to Findler and Felleisen (2002). I first present a contract for sorting functions, which applies both the `list?` and the `sorted?` predicates as a post-condition assertion. I then use contracts to enforce a simple specification for the `first_n` curried function, then `first_n_function` contract in line 5: given a sorting function (according to the contract defined in line 1), a number and a list, `first_n` returns a new list. The detailed behaviour is not specified in the contract, but the returned list contains the first n elements of the given list after sorting.

Our example use of the `first_n` function illustrates why these contracts cannot be immediately resolved. The issue is that we require a contract on the `sort` argument, which must be a `sorting_function`. That is, it must return a sorted list. The function

```
1  (define sorting_function (-> list? (and/c list? sorted?)))
2
3  (define take_n_function (-> number? (-> list? list?)))
4
5  (define first_n_function
6    (-> sorting_function take_n_function))
7
8  (define/contract first_n
9    first_n_function
10   (λ (sort)
11     (λ (n)
12       (λ (l) (take (sort l) n)))))
13
14 (define wrong_sort (λ (x) x))
15
16 (define first_2 ((first_n wrong_sort) 2))
17
18 (first_2 (list 1 2 3 4 5)) ;; This is ok
19 (first_2 (list 5 4 3 2 1)) ;; This must fail
```

Figure 2.2: An example Racket program with higher-order contracts

`wrong_sort` clearly does not fulfill this contract in general (it is just the identity), but if the list was already sorted, the contract is still fulfilled. Contract satisfaction cannot be decided when we call `first_n`, because we produce a new function that has not been called yet. Contract satisfaction still cannot be decided when we define `first_2`, as we haven't given the list to sort yet. When we call `first_2` with a sorted list, all the contracts are fulfilled, so the program should not fail yet. We can only detect the contract violation when we call `first_2` with an unsorted list, as in the example call in the last line, which signals a contract violation.

At the point a contract violation is found, the language must abort computation and report an error location. The contract that failed was that `wrong_sort` was *not* a sorting function, therefore the error report should not point to the current execution point, which is the call to `first_2` on line 19. The problem is not the unsorted argument, but rather that the function that was expected to sort did not. The code where this happened was already executed and is not in the current call stack. In this case, we must connect *post-facto* to the (already completed) call to `first_n` inside the definition of `first_2` on line 16, where we passed `wrong_sort` as an argument to `first_n`. This argument did not comply with the contract for the `sort` argument as specified in line 9 . Note from this example that we had to choose between many possible locations involved in the failure to produce an error. We justified why not to report line 19 in this case, although the line would be a valid candidate if the contract on `first_2` had been a first-order contract instead. I discuss more deeply error diagnostics and the choices an implementation can make in Section 2.2.

2.2 tracking the source of a failure (or a computation)

When a program produces an unexpected result, we want to identify the parts of a program that contributed to producing that result. In this section I discuss two approaches from the literature: program slicing, and blame. Program slicing is connected to provenance data, which I separately discuss.

2.2.1 *Program slicing*

Program slicing is a technique to identify the parts of a program that are relevant to the computation of a particular part of a result. The parts of the result we are interested in are called a *slicing criterion*, and the parts of a program that may affect the criterion are called a *program slice*. From any individual criterion, a program can be sliced to a different program slice.

Weiser (1981) introduced program slicing as a static analysis. He extracted from programs *"a minimal form that still produces the behaviour"*. Working in the context of imperative languages, he chose slices to consist of statements/lines of code, without worrying whether the remaining slice remained a syntactically valid program. Later, Weiser (1982) reported that expert programmers mentally apply slices to reason backwards about the causes of failures, constructing in their minds an abstract representation of the programs they debug. Weiser used this finding to justify putting slicing to practical use. Recently Yoo (2022) integrated static program slicing into IDE tooling, showing that static slicing helps answer reachability questions. However, because these slices are static, by design they must be more conservative and include parts of a program that are not always relevant.

A more precise approach is to analyze a single execution trace to obtain a program slice, which then becomes a *dynamic slice* (Korel and Laski, 1988). Because a dynamic slice needs to account only for a single execution instead of approximating all of them, it can gain precision and reduce the size of the slice, for *"a finer localization of the fault"*. Korel and Laski (1988) define a slicing criteria as a set of program variables and a program counter. They propose that dynamic slices can be *minimal*, although they mention that obtaining minimal slices in general is undecidable. By minimal they mean that no slice with fewer program statements can reproduce the final values of variables of interest (the criterion) using a portion of the original trace. The portion does not need to be continuous: it may skip unnecessary instructions. Zhang, N. Gupta, and R. Gupta (2007) count the lines of code kept by dynamic slices on real open-source software with bugs and validate that dynamic slicing considerably reduces the number of program statements that need to be examined to locate faulty statements. They show that their dynamic slices include between 0.45 and 63.18% of the statements that were executed at least once, which itself represented only a fraction of the total code in programs (.04 to 8.52%). Unfortunately they do not provide specific numbers to compare with static slicing, although they claim those would be larger.

2.2.1.1 *Program slicing on term rewriting systems*

I discuss two approaches from more recent work that formalized program slicing and generalized it to arbitrary languages. In this section I discuss slicing for language workbenches, (via both *origin tracking* and *dependency tracking*), and I discuss slicing for functional languages (via *Galois slicing*) in Section 2.2.2.2.

Language workbenches are programming systems for language designers. Language designers can specify language semantics as *term rewriting systems*: some syntax and a set of rewriting rules. In exchange, the system provides tools based on the language specification, including interpreters,, debuggers, and editing and refactoring services (Erdweg et al., 2015). Most of the work we discuss was based on the ASF+SDF environment (Deursen, Dinesh, and Meulen, 1994).

One of the tools the workbench can provide from term rewriting systems is program slicing. Workbench designers developed a general-purpose slicing system depending on the rewrite rules used to define arbitrary languages. In this context, a "slice" remains defined as the part of a program needed to compute a particular program variable, thus it is still connected to imperative programming. The granularity of the slice, however, is not at the level of program statements, but rather nodes of the program's AST.

To achieve a general-purpose slicing algorithm, the formalism defined two auxiliary relations between the subexpressions in rewrite rules (more specifically, between nodes in the AST of a program on the input and output side of a rewrite rule): the *creation* relation and the *residuation* (or *moving around*) relation. Because these relations directly connect nodes in the ASTs, they do not require workbench designers (nor language designers) to devise a labelling scheme for program sections as blame does (see Section 2.2.3).

Residuation was introduced by van Deursen, Klint, and Tip (1993) under the name of *origin tracking*: a language designer should provide a relation connecting subexpressions in the output side of the rewrite rules with their "origin" in the input side. This relation accounts for the copying and moving of values that happens in a rewrite rule. For example, residuation can track the flow of expressions through a function application. However, residuation is not sufficient to account for new values that may arise during evaluation.

This limitation of residuation led to the introduction of the *creation* relation when Tip (1995) combined origin tracking with *dependency tracking*. The slices obtained with his algorithm do not contain just relevant information, so he introduces a *decluttering* process to remove "unnecessary information" from a slice.

As they formalize this new approach, Field, Ramalingam, and Tip (1995) argue that static and dynamic slicing are two instances of the same *constrained slicing* process. Dynamic slicing refers to constrained slicing of closed terms (with no free identifiers pending to be linked), and static slicing refers to open terms (as rewriting will get stuck in a "compiled" version of the program, waiting for linking). They formally define creation and residuation as binary relations between the nodes of the AST of a program "before" and "after" the application of a rewrite rule.[11]

Later Field and Tip (1998) introduce some metatheory for this slicing framework, providing uniqueness, *soundness*, and *minimality* criteria for the slices generated using the creation and residuation relations:[12] Soundness means that slicing computes slices that reproduce behaviour.[13] Slices are also *minimal* if no slice contains "fewer" function

symbols, as they are still considering variable names as their slicing criteria. These theorems also arise when discussing slicing for functional programs in Section 2.2.2.2.

2.2.1.2 *Program slicing for static type error explanations*

As mentioned in Section 2.1, compiler and type-checker error messages are not always the best. One can use slicing to improve these error messages. Tip and Dinesh (2001) applied program slicing to type checking, where dependency tracking supported *type slicing* as a tool to improve static error reporting.[14] They reused ideas from term rewriting systems by framing type checking as rewriting, where type checking *"rewrites a program's AST to a list of type errors."* Unlike a program slice which preserves execution behaviour, a *type slice* is a projection of the original program that preserves type checking behaviour. In a nutshell, rules rewrite program constructs to their types, and rewritings get specified only for type-correct places: any stuck term left at the end of rewriting signals inconsistencies in the original program. These inconsistencies are then post-processed into a list of type errors.

Tip and Dinesh (2001) discusses some metatheoretical results for type slicing. They claim *soundness* for their type slices.[15] Their slices are not always minimal, but they provide a secondary algorithm to slice out further "irrelevant" parts, and describe designs for rewrite rules that tend to produce less informative slices.[16] This slicing approach does not specifically deal with type inference.

Wand (1986) already observed that the error explanations for type inference failures tended to be poor. He proposed extending the constraint-solving process of type inference to collect slice-related information, *"keeping track of the* reasons *the checkers make deductions about the types of variables"*. Unfortunately, his approach did not provide any metatheory about the properties of these explanations, nor check whether the resulting constraint sets are minimally unsolvable or necessary for the failure.

Error messages in type inference are also poor because implementation details of the type checker, for instance, the order in which type variables get resolved or *unified*, bias error messages and can point developers in the wrong direction. Stuckey, Sulzmann, and Wazny (2003) try to narrow down the source of an error and find *"minimal justifications of type errors"* to explain unexpected types that arise. This process is later refined by the same authors (2004) to improve the messages. Still, they only provide *"Better guesses as to the true source of a type error"*.

Choppella and Haynes (2003) provide a formalism describing unification problems as graphs and use paths in the graph to track source information. Although they mention that these paths can be used to obtain (possible many different) program slices characterizing a unification failure, the paths are the focus of their work and they do not dwell on the connection between paths and program slices. Hage and Heeren (2007) apply "type graphs" representing unification constraints in a Haskell compiler (Helium). They use these graphs to provide candidate fixes in error messages, picking the conflicting type constraint that most likely should be removed based on "expert knowledge" heuristics. Eremondi, Swierstra, and Hage (2019) apply these heuristics in the context of dependently typed languages, combining them with a unification algorithm that is less order dependent and does not stop at the first failure.

Haack and Wells (2004) also suggest applying program slicing to type inference. They argue for type slices, identifying that one of the drawbacks of error messages is that *"blaming just one program node"* causes confusion, as the true problem is finding *"endpoints of a clash between [...] type constructors"*.[17] They identify completeness and minimality criteria for type error slices, where a slice is *complete* if all program points in the slice and the relationships between the program points together guarantee a type error, and it is *minimal* if none of the program points is irrelevant for the type error. They also argue for slicing as the best a compiler can do, which I agree with:

> *"We believe that it is most important to* accurately locate *type errors, and display type error* locations *in a user-friendly way. For* understanding *errors, programmers typically use additional semantic knowledge that cannot be provided automatically anyway."*
> *[Fix locations depend] "on what the programmer has in mind when designing the program. Clearly, a compiler cannot read programmers' minds. Therefore, identifying complete error regions is the best a compiler can do."*
> *(Haack and Wells, 2004)*

2.2.2 *Provenance*

Provenance is information about the origin, history, or lineage of an object (Acar et al., 2013). Provenance also refers to the meta data that a system must collect to be able to answer questions regarding an artifact's history (Carata et al., 2014). Provenance work in databases was the foundation for later work on provenance for programming languages, as ideas about data provenance in databases transfer to programming languages by considering programs as data. In this section, I discuss some of the foundational ideas for provenance first in the context of databases (where they arose) and then how they transfer to functional programming languages. I also describe some of the software tools introduced in the literature that collect provenance information at the language level.

2.2.2.1 *Foundational ideas of provenance for databases*

Executing a database query produces a table of results. Users may be interested in asking questions relating the result obtained to parts of the original database the results came from, as the connections may not be immediately visible. Although the focus of our work is outside the scope of database foundations, the work on provenance for functional languages draws heavily from the ideas in database provenance: the connection between a database and a query output mirrors the connection between a program and its evaluation output, which is our interest. The definitions of provenance for databases mirror the definitions of provenance for programming languages, as we see in Table 2.2.

Cheney, Chiticariu, and W.-C. Tan (2009) survey three notions of provenance in the context of databases. Each notion of provenance collects data to answer, after a database relational query has produced an output table, questions about a portion of the output:

- (Why-provenance: Cui, Widom, and Wiener (2000)) *Why is this tuple in the query answer?* More specifically, what portion of the database contributed to the construction of the query answer? Answers to this question are a subset of the rows in the

database's tables, called a *witness*, which should be sufficient to reproduce the same answer for the same query.

- (Where-provenance: Buneman, Khanna, and W.-c. Tan (2001)) *Where in the database did this part of the answer come from?* More specifically, from what exact place in the database did we copy each field of each row in the answer? Answers to this question connect each data item in the query answer with the places in the database that affect it.

- (How-provenance: T. J. Green, Karvounarakis, and Tannen (2007)) *How was this tuple in the query answer computed?* More specifically, from what combination of rows from the database could we obtain the result of interest, and how many times did we use each of those rows?

We can ask these questions about the complete query answer or about a portion of it. Note that these questions are listed in order of increasing information detail, as it is usually the case that once we set on answering one of these more detailed provenance questions, we can indirectly also answer the less detailed kinds presented before.

These notions of provenance in databases are formalized via *provenance semirings*. I spend the rest of this section explaining the intuitions behind provenance semirings, as the formal notions of provenance in databases can be used to understand the work on programming language provenance and vice-versa.

T. J. Green, Karvounarakis, and Tannen (2007) introduce provenance semirings by noting that provenance propagation in relational databases can be mathematically encoded as an algebra which they name a *positive algebra*. This positive algebra is equipped with two operations that form a semiring structure,[18] thus the name *provenance semirings*. This positive algebra mirrors the relational algebra that encodes the process of computing a database query, following the structure of a query definition to aggregate and propagate information. The authors in the paper model why-provenance and how-provenance as examples of different operations that, when combined with the positive algebra, collect sufficient provenance information about the query at hand. For why-provenance, they propagate a set of tuple ids via set union. For how-provenance, they introduce a notion of *polynomials* associated with each tuple in the result of our query, which are propagated via polynomial algebraic operations.

These polynomials are only a choice of "convenient" syntax by the authors, which may be more confusing than helpful. The authors rely on our ability to expand a polynomial like $2 * t_1^2 + t_2 * t_3$ as $(t_1 * t_1) + (t_1 * t_1) + (t_2 * t_3)$, and to remember the fact that at this point $*$ and $+$ are just symbols representing abstract operations, and not the numerical operations we usually associate with them.

Although mathematically elegant, these polynomials can obscure their intuitive meaning. For example, when a tuple in the result query is assigned the polynomial

$$2 * t_1^2 + t_2 * t_3$$

we are expected to see that this polynomial represents three ways of computing the tuple: two that use t_1 twice, and one that uses t_2 once and t_3 once, as the symbolic variables represent ids of tuples in the original database.

Representing this information as a collection of lists of tuple identifiers instead of a polynomial, though semantically equivalent, makes the intuition clearer. The polynomial in our example could also be equivalently represented as the list of lists `[[t1, t1], [t1, t1], [t2, t3]]`, which makes the three ways of computing the result explicit. Each list in the collection contains all the tuples from the original database involved in one way of computing the result in question, and the collection contains as many lists as there are ways to compute the result from tuples in the original database. As each way is represented with a list and not a set, it can also remember how many times each tuple from the original database was used.

2.2.2.2 *Provenance for execution traces*

In the functional programming community, the first formal model of language-level provenance is the Galois slicing approach by Perera, Acar, et al. (2012).[19] This work defines a query language to ask, after a program in the *Transparent ML* language has been evaluated to a value, two possible questions about a portion of the output:

- From what portion of the original program did the interpreter compute this portion of the output?
- What is the portion of the execution trace that was involved in computing this portion of the output?

For example, consider the following program:[20]

```
nth 1 (3 :: 5 :: [7, 11])
```

This program produces `5` as an output. The authors provide an instrumented runtime semantics that collects provenance information while evaluating the program. We can then ask from what portion of the original program did the interpreter compute this output, to obtain the following slice:

```
nth 1 (□ :: 5 :: □)
```

Both the semantics and the query language rely heavily on the notion of a *program slice*. In this context, a program slice is a program which can have *holes* (□) in place of subexpressions. Intuitively, replacing a subexpression with a hole represents the idea of pruning the program AST, discarding the subtrees that are not of interest. The main idea is that program slices characterize sub-parts of a program, just like we could collect some table rows to characterize sub-parts of a database and form a witness.

We can also use the notion of a program slice to ask provenance questions about a portion of the result. For example, if a program produced as output a list of values like `[1, 2, 3]`, we could use the slice `[□, 2, □]` to ask for the portion of the original program from which the interpreter computed the second argument of the list. Using slices is necessary because the output value is a program expression, which are structured differently than the output of a database query. As the output of a database query is a table, one can ask provenance questions by selecting the tuples of interest among all the tuples in the output. For program outputs, we take slices.

There are many similarities between the process of Galois slicing in functional programming languages and provenance semirings in databases. Table 2.2 compares both

	How-provenance in positive algebras or provenance semirings (T. J. Green, Karvounarakis, and Tannen, 2007)	Functional Programs that Explain their Work (Perera, Acar, et al., 2012)
System process without provenance	From *a database instance and a relational database query*, use the *relational algebra* to obtain a *query result table*.	From *a program*, use the *evaluation semantics* to obtain an *evaluation result*.
Collection of provenance information	From the same database instance and relational database query, use the *positive algebra* to obtain a *provenance polynomial table*, mapping result table tuples to their polynomials.	From the same program, use the *tracing semantics* to obtain a *provenance trace*, which encodes the complete evaluation process.
Formulating a provenance question	From the query result table, extract a *tuple subset* of interest.	From the evaluation result, extract a *slice* of interest.
Computing the provenance answer	From the provenance polynomial table, extract the *polynomials* of interest via filtering over the tuples of interest.	From the provenance trace, extract the *least trace slice* of interest via *trace slicing*
Interpreting the provenance answer	Over the list of polynomials, *you can observe all the ways to obtain the tuples of interest*. Considering polynomials as a collection of lists, *if you pick a list per result, and combine the tuples of interest into a database instance (witness) and perform the same query*, you will get a result tuple that still contains the results of interest as a subset.	Over the least trace slice, *you can use the derivation tree of the "uninterp" relation to observe the complete evaluation process leading to the portion of interest of the program*. The "uninterp" relation connects a value slice to a program slice through the trace. *If you run this slice as a program again*, you will get a resulting value that still contains the slice of interest as a prefix.

Table 2.2: A comparison of the process required to obtain and use provenance in databases (with semirings) and in functional programs (with slices)

systems at a high level: using provenance semirings to obtain some how-provenance and Galois slicing to obtain provenance over functional programs. The goal of this table is to draw intuitive conceptual connections between both systems.

The Transparent ML language is also used by Acar et al. (2013) to analyze different kinds of provenance information that can be extracted from traces, also providing some security analysis over the information disclosed by these language provenance systems. Ricciotti et al. (2017) extend the slicing framework to a more complex language that supports mutable state and exceptions, using abstract interpretation to deal with the increased complications of mutable state.

I apply these ideas of slicing and tracing program executions to the problem of run-time type information and error reporting for gradually typed languages. This work has to be adapted, because the provenance framework by Perera, Acar, et al. (2012) does not deal with run-time type information, but rather with the evaluation semantics of expressions. We address two issues to adapt their work to our context.

A first issue is that it is not immediately obvious how to encode the changes in precision of the type information that happen as the program runs. This information is at the core of the runtime semantics of an AGT-based gradual language, and thus it is necessary to encode this information to generate a program slice. It is also not immediately obvious how to use this information to generate a program slice that preserves a minimal amount of information from the original program and still generates the run-time type inconsistency, so that it could be used for debugging.

A second issue is that the formalism for tracing by Perera, Acar, et al. (2012) relies heavily on the style of its evaluation semantics formalism (a *big-step semantics*), which most formalizations of gradually typed languages do not follow. In particular, the Abstracting Gradual Typing framework that I am interested in extending uses a different formalism for semantics (in particular *small-step semantics*). Adapting these provenance ideas to run-time type information and other formalisms for evaluation semantics is a fundamental part of this research project.

This is not the first work to connect run-time error reporting with provenance. Dimoulas (2012) characterizes the labels in a contract blame semantics as a means of tracking ownership, and therefore as a form of provenance. However, because a blame semantics explicitly chooses to drop labels as evaluation progresses, the provenance data is by no means a complete representation of the elements involved in computing the failure. It also assumes a single pair of locations involved in the failure as the blame report and discards the information of some locations, as discussed in Section 2.2.3. A blame semantics could be considered as only a limited form of provenance, because it does not produce a notion of witness from which we can reproduce the failure: a blame report would mark a portion of the program as involved in the failure, but not as sufficient to reproduce the issue.[21] Dimoulas' approach could intuitively approximate a why-provenance system, but no connection has been proven formally.

2.2.2.3 *Software tools to support programming language-level provenance*

To understand execution *failures* one can ask *why* questions about run-time information (e.g. "Why is the value of x greater than 5?"), or *why not* questions (e.g. "Why did this

specific program instruction never execute?"). Several tools have been developed to deal with why and why not questions:

- Ko and Myers (2004, 2010) develop a debugging tool for Java called the *Whyline* that lets users ask provenance questions about execution traces (for example, why did and why didn't some event happen). Both of these questions are answered in terms of an execution trace. When answering negative questions about instructions, they provide some guidelines for the reason (an exception was thrown, method was never called, etc.). However, the authors do not formalize the system or provide guarantees of correctness, and they recognize that negative questions in their system can be insufficiently precise because of performance concerns.

- Wu et al. (2014) apply provenance in their *Y!* system to identify *negative events* in distributed systems: for example, the absence of a required network packet. They do so by examining all possible causes that could have produced the missing effect with standard provenance and show the reason why each of them did not come to pass. They refer to this counterfactual explanation as *negative provenance*. They only apply this idea in the context of networks.

- Nelson et al. (2017) explore answering both why and why not questions in the context of the *Alloy* model-finder, which must also deal with under-constrained specifications and multiple explanations. They provide a formal definition of provenance in their model, but it is not clear how to translate their ideas from model-finder to language implementations.

- Ikeda et al. (2012) annotate database oriented workflows generated with their *Panda* tool, recording provenance over the scripts and database queries used, as well as providing visualizations. Although the authors claim their tool is useful for debugging, they do not explicitly accept negative queries and rely on the expertise of the user to identify irregularities in the output and analyze the provenance information of incorrect outputs.

- Johnson et al. (2015) developed the *Pidgin* program analysis tool, which collects program dependence graphs that capture information flows in JDK-based programs. They also design a query language that can be used to query the information flow graphs regarding whether the program guarantees safety properties like noninterference. Although not explicitly named as a provenance system, the query language is expressive enough to allow users to perform provenance queries through the Pidgin Query Language, and can be used to identify counterexamples to expected system properties and thus can be used for debugging. However, like in the case of Panda, we cannot use Pidgin to answer "why not" questions.

- The *noWorkflow* system introduced by Pimentel et al. (2017) modifies the semantics of a Python interpreter to collect provenance information in the context of scientific experimentation. Their goal is to facilitate experiment reproducibility rather than debugging. To achieve this goal, the system identifies individual executions with trial numbers and collects provenance of Python scripts. The information collected

includes the locations of function definitions used in the script, deployment environment information like the versions of libraries, and the I/O context used in each particular run (including files read and any file modifications done through the run).

- ProvBuild (Hu et al., 2020) builds on top of noWorkflow to provide a debugging tool for Python scripts. From an execution trace (which collects provenance information) and a function or variable name (the slicing criterion), it provides an executable program script with all the dependencies from the trace (a program slice). The interface is quite flexible: the slice can be modified and the tool can merge the changes into the original script. This work shows that a run-time slicing tool can be useful for debugging.

2.2.3 *Error diagnostics for higher-order contracts: the origins of blame*

When we introduce contracts in Section 2.1.2, we mention that contract violations may choose a place in the code to report in the error message, but we do not specify how to choose these places. The first approach by Findler and Felleisen (2002) uses *obligation expressions*, run-time contract checking expressions that carry references to two code locations. These locations are both the caller and the definition of the callee for a contract check, and are used to assign "blame" when contracts do not hold. *Blame* is just a kind of program result, an error message pointing towards a code location. Intuitively, the location reported with a blame expression is assigned as the source of the run-time failure. For example, when running the program in Figure 2.2, the Racket contract implementation assigns labels to each code location that introduces a contract. A contract fails in line 19, and Racket identifies that the failing contract comes from line 9: the call to `sort` in line 12 did not produce a `sorted?` list, which was the expected result of a `sorting_function`, which was the expected argument of a `first_n_function`.

However, having an implementation does not tell us if these error messages are "right", or even *what* makes an error message right. The original work on higher-order blame made no claims of correctness for these error reports. Previous work on object-oriented contracts (Findler and Felleisen, 2001) provided formal contract soundness guarantees when all the contracts pass, but made no claims about where a failure should point when found. Since then, different works have attempted to answer these two questions for higher-order contracts with increasing precision, as each work identifies flaws in previous results. Blume and McAllester (2006) made the first claims about the semantics of higher-order contracts. They found that the original semantics of Findler and Felleisen (2002) could be given a formal contract semantics under some reasonable (though extra) constraints on what a contract can be: allowing arbitrary contract code is too flexible, as we can declare contracts that diverge or that produce themselves errors instead of acting as specification predicates. They restricted the code that can be used in contracts to be functions and provided a semantics guaranteeing that *"when the algorithm blames a contract declaration, that contract declaration is actually wrong"* with respect to their semantics. Findler and Blume (2006) discuss the difference between both approaches with respect to the meaning of the *weakest* contract and who gets blamed in the case where values flow through weaker contracts.

When an obligation expression evaluates to an error, the error can come only from the code over which pre- and post- conditions are checked. This approach was found again to be insufficient: in more elaborate contracts like the dependent contracts introduced by Findler and Felleisen (2002), (that is, contracts that may refer to the specific arguments of a function call on a post-condition), it might be the case that the *contract itself* is incorrect and must be blamed. Dimoulas and Felleisen (2011) rebranded the expressions used to check the contracts at run time as *monitors*, and provide them with a new semantics. Their semantics blames the contract itself when post-conditions refer to the arguments of the call in inconsistent ways.[22] To justify their run-time check semantics, Dimoulas, Findler, et al. (2011) introduced a notion of *blame correctness*,[23] which specifies the valid sources from which an error report label may arise. Language designers could then establish that a run-time check semantics is "correct" by proving its corresponding theorem. The authors rapidly recognized that the notion of blame correctness proposed as a property was unfortunately insufficient: semantics that do not check the internal consistency of a contract are still "blame-correct". Dimoulas, Tobin-Hochstadt, and Felleisen (2012) introduce a more stringent criterion, *complete monitoring*: a complete monitoring semantics in its original form subsumes "blame correctness", but also guarantees that a failing monitor does indeed produce an error, something not required by blame correctness, where evaluation of the dependent post-condition contract could diverge. We provide more background on complete monitors in Section 2.2.4.

But the refinements above do not tell us whether a blame report points to the right place for a developer to look. For example, consider again the Racket program from Figure 2.2. The Racket blame semantics satisfies complete monitoring, and when the program runs, an error message reports that the contract in line 9 failed. The argument given to `first_n` was not a `sorting_function`. Still, nothing in the error report provided by Racket connects the failure with the incorrect construction of this particular `first_n_function`, neither the binding of `wrong_sort` in line 16 nor its definition in line 14 appear anywhere: the programmer must reconstruct this extra information on their own from other bits and pieces.

More generally, these refinements do not answer whether blame helps developers find the causes of run-time errors. Lazarek, King, et al. (2019) argue that people should use a blame system by applying a methodology they call *blame shifting*. According to this methodology, a developer is expected to be able to inspect a code portion that is blamed and recognize when the code is correct in spite of the error and diagnose the blame assignment as incorrect. The developer is then expected to impose more precise contracts in the program, forcing the system to assign blame somewhere else. Fixing errors thus becomes a process of increasing specification to avoid being blamed, at the cost of first (possibly over-) committing to further contract obligations.

Lazarek, King, et al. (2019) also propose an experimental approach using mutation testing to evaluate whether blame shifting provides two outcomes desired from blame in the folklore.[24] First, the *blame trail* property asks whether programmers that heed blame and focus their attention to blamed components eventually locate the faulty component.[25] Second, the *search progress* property asks whether programmers keep getting "closer" to uncovering the location of the bug when they rely on using a blame shifting strategy for

guidance.[26] They find that these properties tend to hold *most of the time*, but *not always*, finding counterexamples for each.[27]

Their research identified several limitations with blame, even before addressing the more general limitations we discuss in Section 2.1. Blame expects a debugging approach that requires increasing specification commitment, and cannot be taken as a guaranteed approach to find the inconsistency that provoked a failure. We discuss further limitations of blame in Section 2.5.3.

2.2.4 *Complete Monitors*

Complete monitoring recognizes that programs are a combination of *parties*, that is, components (like modules) where values and expressions originate from or traverse through. Each party can make assumptions about the other parties involved in the program execution, and the assumptions each party makes can sometimes be incompatible with each other. A language is a complete monitor if the flow of values from one party to another is always guarded with a run-time check that eventually ensures that any assumptions made are consistent with reality. These monitors act as a boundary crossing where responsibility for assumptions can be transferred: under complete monitoring, no illegal boundary crossings between parties are allowed.[28]

To provide this guarantee, the complete monitoring theorem relies on a runtime semantics that preserves information about the flow of values through evaluation. Expressions and values collect the history of the parties involved in their computation. These parties are called *owners* and are represented as a stack of labels.

Labels can only be forgotten or discharged under very specific circumstances. A logical judgment over programs called the *single-ownership discipline*[29] forces that transfers of ownership can only happen through explicit boundary crossings, the monitors. Monitors are wrappers for code that can delay checking properties until the point where values need unwrapping, and are intended to act as checked boundary crossings that may transfer ownership.

The core of complete monitoring consists of the following two ideas:

- *Preserving party information*. Evaluation preserves all ownership information by default. One may discard party information only when a party stops being involved in a computation. For example, new first-order values resulting from a call to a primitive function don't need to carry the history of owners of the arguments to the call, the information of a non-followed branch in an if statement can be discarded, and substituting a value for an identifier that does not occur in the program can safely forget the parties involved in computing the value that is being vacuously substituted.

- *Preserving a single-ownership discipline*. There can be many owners in the same program. In particular, programmers are free to add as many ownership annotations to expressions as they want, provided that monitors protect every change of owner. The single-ownership discipline does not enforce a single owner, but it enforces that *ownership changes can only happen through explicit boundary crossings*. Thus annotations must be consistent with the checks performed at boundary crossings.

Single-ownership forces constraints on the semantics. To preserve single-ownership, an explicit boundary can be removed only when the owner on both sides is the same, when the monitor check removes labels, or when the monitor errors out. Single-ownership does not imply that values cannot flow from one owner to another, but it does imply that all ownership transfers are explicit, and that the previous owner may be forgotten only under limited conditions.

This discussion leads me to the following definition for complete monitoring:

Definition 1 (Complete monitoring)**.** *A runtime semantics is a complete monitor if the runtime semantics preserves a single-ownership discipline, while at the same time trying as much as possible to break that single-ownership discipline by preserving all ownership information throughout evaluation outside the specific case of monitor expressions.*

We can think of a complete monitoring semantics as succeeding in preserving single-ownership in spite of its own antagonistic behaviour. The combination of a single-ownership discipline with antagonistic ownership information preservation constrains the semantics of monitor wrappers and their handling of ownership information. Monitors are intended to be removed only when the properties that the monitor is supposed to enforce are guaranteed to hold, thus the values they wrap can be freely transferred to a different owner.

Note that the definition I provide for complete monitoring does not directly refer to labels. The labels (and labelling requirements) of complete monitors are only a metatheoretic artifact: they do not need to exist in the language implementation, but the language must be instrumented[30] to include labels and propagate them as part of the proof of the complete monitoring theorem. The rest of the instrumented reduction relation cannot be affected by the introduced labels: the original semantics should be recovered by erasing the labels so that the original semantics preserves the property of interest.

Our focus on the core ideas of complete monitoring lets us disregard details that normally appear in discussions of complete monitoring but are not fundamental to the theorem and the property of interest:

- Runtime semantics in complete monitoring papers tend to reverse lists of owners when evaluating function calls, in order to preserve a connection to the subtyping contravariance that happens on function arguments. However, reversing labels is orthogonal to complete monitoring *per se*, because the theorem holds with or without this reversing.

- That evaluating primitives reduces the set of owners to one owner is an optimization. A complete monitor could still preserve all party information when evaluating primitives, without affecting the property of interest (which however may become harder to prove).

- Although the standard complete monitoring theorems reference blame, complete *monitoring* is intended only as a property of boundary checking. Complete monitoring does not impose direct constraints over the error messages produced by blame. Complete monitoring only guarantees that the language contains sufficient checks, not what kind of error messaging and reporting the language provides, and the connections to blame are just to reuse its labelling infrastructure for the proof.

The idea of *ownership* was introduced as part of blame correctness by Dimoulas, Findler, et al. (2011). Dimoulas, Tobin-Hochstadt, and Felleisen (2012) reuse this notion of *ownership* in their blame semantics to address limitations in "correct blame", which couldn't be used to distinguish two semantics and determine whether the validity of the contract itself was checked correctly. One of the semantics they explored never assigns blame to contracts themselves, but is still blame-correct. The authors proposed *complete monitoring* as a stronger property that could be used to distinguish between these semantics.

Blame and complete monitors have not only been applied to behavioural contracts, but also to gradual typing. Before discussing the connections between both contexts, we have to introduce gradual typing.

2.3 GRADUAL TYPING

In the past, programming languages were classified as either statically typed or dynamically typed, as described in Section 2.1.1. These languages provide disjoint benefits: statically typed languages can guarantee properties before running (and shipping) a program, and dynamically typed languages permit rapid prototyping when invariants may not yet be agreed upon. However, both benefits have their place in the development process. It would be preferable to not be forced to choose only *one* of these benefits *for the whole development process*, but instead to combine both benefits in the same language.

2.3.1 *Prehistory*

The formalization of combining static and dynamic typing in the same language dates back at least to the late eighties. Abadi et al. (1989) introduced a special type `Dynamic` for dynamically typed values, as well as a `typecase` construct to pattern-match dynamic values, later implemented in the CAML language by Leroy and Mauny (1991). Thatte (1989) proposed *quasi-static typing*, where any statically typed value could also have type `Dynamic` via subtyping.

Cartwright and Fagan (1991) introduced *soft typing*, an approach to translate dynamically typed programs into a statically typed intermediate language. Soft typing infers types first, and wherever type inference would have produced a static error, soft typing instead inserts run-time checks. In this approach, programs are *never* rejected statically, all failures happen at run time.

Anderson and Drossopoulou (2003) introduced a notion of incremental typing, with a nominal type system for a formal calculus in the style of JavaScript called BabyJ. In that language, the runtime semantics of the typed language is defined by erasing types and running programs in the dynamically typed language, an approach much in the style of optional typing, which we describe next.

Bracha (2004) proposed naming languages that accept programs missing some type information as *optionally typed*.[31] Although optionally typed languages claim the advantages of combining static and dynamic checking in the same language, they allow programs missing any type information to live dangerously: once we run these programs, any of the static assumptions could have been broken.

Optionally typed languages claim to have the "advantage" of no run-time overhead to check invariants, but this "advantage" comes at a steep cost: programmers cannot rely on any type invariants until the program is completely typed.[32] If programmers want a guarantee that a type invariant holds, they must sprinkle run-time checks manually, but receive no guarantee from the system that the run-time checks they introduced are sufficient. This approach has unfortunately seen widespread adoption in industrial languages like Google's Dart,[33] Microsoft's TypeScript,[34] and Facebook's Hack.[35]

2.3.2 *Beginnings*

It took almost two decades for a system to appear in the literature in which static and dynamic code can interact seamlessly and where the static assumptions are preserved throughout a program execution. Then two appeared in the same year: gradual typing by Siek and Taha (2006) and migratory typing by Tobin-Hochstadt and Felleisen (2006). In gradual typing, interactions may happen at the granularity of expressions.[36] In migratory typing, interactions may happen only at module boundaries, thus program modules are either completely dynamically typed or completely statically typed.[37] I follow the first approach, where developers can choose, at expression-level granularity, whether to use static or dynamic checking. This approach has no loss of expressiveness as module-level granularity can be considered a special case. The latter approach is applied in practice in the Typed Racket language.[38]

Gradually typed languages permit the combination of statically checked and dynamically checked properties in the same program. Programmers can have the rapid prototyping benefits of a dynamically typed language and achieve the safety guarantees of a statically typed language. The programmer controls which parts of a program have statically checked properties and which properties are checked at run time.

Gradual typing began with the gradually typed lambda calculus (GTLC), a simple language with first-class functions and mutable references.[39] Following Anderson and Drossopoulou (2003), Siek and Taha (2006) introduced a type representing missing type information, the *unknown* type (denoted ?). The unknown type can be used to construct types with partial missing information. For example, a type $? \to ?$ represents a 1-argument function, but provides no information about the type of its argument or its return type.

To use these gradual types, the type system replaces type equality checks with the notion of *type consistency*, an approximation of type equality that accounts for missing type information. Type consistency implies an optimistic assumption, that types missing information *could* be equal.

If a language simply loosens the type system to accept more programs, it also loses the formal guarantees from the original static language. The definition of type consistency that Siek and Taha proposed allows their type system to accept programs that may perform undesirable operations at run time. For example, the following syntax represents an anonymous function in GTLC.

$$(\lambda(f{:}?)\,.\,f(5))$$

This function receives a callback function f as an argument and simply calls its argument with the parameter 5. We write the function explicitly with no static type information about its argument by using the unknown type. The type system cannot statically detect incorrect applications of this function, for instance applying it to a Boolean argument instead of to a function that expects numbers.

The key contribution by Siek and Taha (2006) was to add *run-time checks* to ensure that the optimistic assumptions made when type checking the program do hold in practice. Siek and Taha proposed a type-directed translation that introduces run-time checks in the form of *type casts*. These checks are needed when a property cannot be guaranteed to hold statically because of missing type information. For example, the translation introduces a check that f is a function before applying it to 5. In our example, if f is the Boolean `false`, the type cast fails and interrupts execution, instead of possibly crashing or getting stuck.

The program must have these run-time checks because type consistency, our relation of interest, is not transitive and thus types that are indirectly consistent may become inconsistent as computation progresses. When our example function is called with `false` as an argument, the type checker accepts the program because the type Boolean of `false` is consistent with the unknown type ?, and the unknown type of f is consistent with the function type $? \to ?$. But the Boolean type is not consistent with any function type, which is why the run-time check fails.

2.3.3 *What's gradual typing, anyway?*

As the name "gradual typing" became popular, many languages claimed the "gradual" title. Although Siek and Taha specified formally that their language was *gradual* because of their *consistency relation*, it was not always clear how to adapt the definition to more advanced type disciplines. The word "gradual" was used sometimes to also refer to systems with optional typing, like TypeScript and Perl 6. Languages following a migratory typing style are also called gradual. Politz, Quay-de la Vallee, and Krishnamurthi (2012) proposed "progressive types", an alternative to gradual typing in the shape of a type-and-effect system,[40] so progressive types also sometimes get identified as gradual. Even *among* gradual languages, different run-time checking strategies arose to deal with different kinds of issues. If the semantics are different, are all of them actually gradual? There was no clear way to answer. This section explores several criteria that have appeared in the literature to justify whether a language is "gradual" or not.

2.3.3.1 *What do programmers want?*

Some researchers sought to make sense of this situation by asking programmers for their expectations of a gradual language. Tunnell Wilson et al. (2018) found that the developers in their sample expect type annotations to mean something: they should impact evaluation. Thus they dislike optional-like semantics, and expect both data- and control-flow enforcement of type invariants, as intended in gradual typing. Hoeflich, Findler, and Serrano (2022) checked for type inconsistencies in "DefinitelyTyped", a repository of type annotations for JavaScript modules to use them in TypeScript programs,

and provided pull requests to correct the inconsistencies. In contrast to the findings by Tunnell Wilson et al. (2018), they found that sometimes *"programmers preferred incorrect types"*, either due to encapsulation choices (they didn't want to extend interfaces with extra fields) or for backwards compatibility with code that depends on those annotations (changing the types would stop existing code from type checking).

2.3.3.2 *Many semantic options*

GTLC is a simple language, thus further research was required to answer whether the ideas underpinning gradual typing can be directly applied to languages with more complex type systems and language features. These languages rely on concepts like subtyping, which are more elaborate than type equality, to determine whether a program should be accepted as valid or rejected. The question of gradual typing's broader applicability generated a decade-long effort in the community to build *ad hoc* gradual versions of increasingly complex languages: Siek and Taha (2007) for records, objects, and subtyping, Ahmed, Findler, et al. (2011) for parametric polymorphism, Takikawa, Strickland, et al. (2012) for first-class classes, Sergey and Clarke (2012) for ownership types, Jafery and Dunfield (2017) for refinement types, Toro, Garcia, and Tanter (2018) for secure information flow, and my own work for effect systems (Bañados Schwerter, Garcia, and Tanter, 2014, 2016).

But even for the simpler languages, what the semantics "should" be was not always clear. Besides the original semantics of Siek and Taha (2006), later referred to as "Natural", and the "optional" approach of disregarding type info altogether (called "Erasure" for comparison purposes), several other semantics for gradual typing have appeared, dealing with different semantic issues.

One issue arises when language designers have limited control over external dynamically typed libraries or language runtimes and thus cannot instrument them with sufficient checks. Vitousek, Swords, and Siek (2017) designed a semantics where sufficient checks are guaranteed only for the values under *the designers'* control. This semantics is called *Transient*.

Another issue is that run-time structural type checks can be costly. One way to avoid the issue altogether is to restrict the gradual types to a nominal type system, as names can be cheaply checked for equality. A semantics that does not perform structural checking is called *Concrete*, first applied to the "Thorn" language by Wrigstad et al. (2010),[41] and then to a variety of other languages (Lu et al., 2023; Muehlboeck and Tate, 2017; Richards, Nardelli, and Vitek, 2015). As structural checks are forbidden, language semantics following the concrete approach cannot be gradual over higher-order function types or any types whose checks cannot be immediately resolved.

Because they are so different from one another, these languages and approaches are hard to compare. Greenman, Dimoulas, and Felleisen (2023) provide a framework for an apples-to-apples comparison of different run-time type-enforcement strategies: Natural, Transient, and Erasure from the literature, plus new theoretical ones, and list several comparison criteria:

- Type safety
- The "gradual guarantee", which we introduce next

- Complete monitoring
- Blame soundness and completeness (connected to blame correctness)
- Whether a semantics produces more or fewer run-time errors
- Whether values are wrapped in delayed checks or not

We use this list as a basis for our discussion, though we also introduce other criteria from recent literature.

2.3.3.3 *The current bare minimum: gradual guarantee and other refined criteria*

Although Siek and Taha (2006) introduced gradual typing, the specification of what makes a language gradual was made more precise later by Siek, Vitousek, et al. (2015) in their *refined criteria*, all of which apply to the GTLC language. First, the language must be a superset of a static and a dynamic language. "Being a superset" here means that fully annotated terms are equivalent in the gradual and the static language, and dynamically typed programs can be embedded in the language. Second, the language must be type safe and a *blame-subtyping theorem* should hold.[42] Third, a set of theorems should hold, called the *gradual guarantee*.

The gradual guarantee theorems state that the following two properties must hold for any well-typed program:

Definition 2 (Static Gradual Guarantee)**.** *Making the type annotations in a well-typed program more imprecise implies the program remains well-typed, at a (possibly) more imprecise type.*

The static gradual guarantee implies that removing type restrictions should not stop a program from type checking. This property means that any program that is allowed to run in the system will still run if any of the type commitments in it are weakened.

Definition 3 (Dynamic Gradual Guarantee)**.** *Take two well-typed programs that only differ with respect to the precision of their type annotations:*

1. *If the* less *imprecise program evaluates to a value, the* more *imprecise also evaluates to a (possibly) more imprecise version of that value.*
2. *If the* less *imprecise program diverges, the* more *imprecise must also diverge.*
3. *If the* more *imprecise program evaluates to a value, the* less *imprecise either evaluates to a (possibly) less imprecise version of that value, or it evaluates to a run-time type error.*
4. *If the* more *imprecise program diverges, the* less *imprecise either diverges or it evaluates to a run-time type error.*

As observed in Greenman, Dimoulas, and Felleisen (2023), even optionally typed languages may fulfill the dynamic gradual guarantee: when type annotations do not affect the runtime, annotation changes are no-ops and the guarantee trivially holds. What is then surprising is that the gradual guarantee may actually break at all: some languages provide properties stronger than type safety, like parametricity. The sealing of terms required to guarantee parametricity breaks the gradual guarantee. The interactions between parametricity and the gradual guarantee have sparked intense scrutiny and design refinements to satisfy both properties in one language.[43]

2.3.3.4 *The "contrapositive" key of the gradual guarantee*

The gradual guarantee was designed to imply consequences for run-time type errors. With the gradual guarantee, if a program accepted by the type system produces a run-time type error, there exists an inconsistency among the static invariants *already declared* in the program, an inconsistency that unfortunately could not be identified by the static type checker. Therefore, we cannot fix the error by adding *more* type information, but only by either removing annotations (making the program more dynamic) or by changing the existing type constraints in the program.

But the theorem makes no direct claims guaranteeing behaviour when we start from a run-time type error, so to achieve this error reasoning principle with the gradual guarantee we need to read the theorem in a "contrapositive" fashion:

Definition 4 ("Contrapositive" gradual guarantee)**.** *If a program evaluates to a run-time type error, increasing type precision of the program also evaluates to a run-time type error.*[44]

We placed "contrapositive" in quotes, as it is not always possible to deduce this separate property from the original definition of the gradual guarantee. Only *if* the language is deterministic *and* type safe,[45] *then* one can imply this new property via contrapositive arguments. Otherwise, we must prove this property separately.

The combination of the gradual guarantee and its contrapositive version has also been explored by New (New and Ahmed, 2018; New, Licata, and Ahmed, 2019), although described via a stronger property, *graduality*, which implies the dynamic gradual guarantee:

Definition 5 (Graduality)**.** *For any (open) well-typed term, any "less dynamic" version of the term observationally approximates it up to error. More formally, consider any possible valid context against which we could link a program, and any two precision-related terms to link the context with. Graduality holds if for any possible choice of the three, when the more imprecise linked program does not produce an error and terminates (diverges), then the more precise linked program either also terminates (diverges) or fails with a run-time error.*[46]

New, Jamner, and Ahmed (2020) explain graduality as a formalization for the intuition that *"making types more precise should not impact the* partial *correctness of the program itself"*, the same intuition as that of the dynamic gradual guarantee. *"This means that a programmer can add types to a portion of their program and know that the program as a whole still operates the same way, unless a new dynamic error is raised, in which case there is a flaw either in the code or in the new annotation that was introduced."*

2.3.3.5 *Other criteria*

Besides these refined criteria, other approaches to classify languages as "gradual" or to compare semantics have appeared in the literature.

- Greenman and Felleisen (2018) introduce a notion of "Err-approximation", as a way to categorize runtime semantics of gradual typing. Different runtime semantics are usually quite similar, the only difference being that one semantics produces more "errors": programs that evaluate successfully in one semantics may not do

so in another, because the latter is stricter about detecting inconsistencies between the type annotations in a program. Greenman, Dimoulas, and Felleisen (2023) use this criterion (among others) to compare languages, calling it an "error preorder" among semantics: *"The error preorder compares the relative permissiveness of types in two semantics. Natural accepts the fewest programs without raising a run-time type mismatch and Erasure accepts the greatest number of programs."*

- Greenberg (2019) distinguishes between two lines of gradual typing: "dynamic-first", aiming to account for dynamic idioms and evolve the design accordingly in the style of migratory typing, and "static-first", relaxing a statically typed language to account for dynamism. The paper claims that most theory-based work on gradual typing takes a static-first approach and most implementations of gradual languages with practical intent take a dynamic-first approach. He summarizes gradual typing as *"finding runtime-enforceable safety properties that simultaneously (a) allow one to relax the strictures of type checking in part of one's program while (b) not compromising the safety guarantees in the checked parts of the program"*.

- Garcia and Tanter (2020) identify that most gradual typing work relies upon type *safety*, which sometimes forsakes type *soundness*, a stricter semantic criterion they hope to establish for gradually typed languages.

- Jacobs, Timany, and Devriese (2021) start from the soundness criterion of Garcia and Tanter (2020) to argue for a new criterion for a "good" gradual language: that the embedding from the static to the gradual language should be fully abstract (i.e., preserve *all* source language observational equivalences). They show that some "erroneous" gradualizations that nevertheless satisfy the refined criteria are rejected by this criterion, and that the criterion holds for a gradualization of STLC plus equirecursive types. Formally, this requires that the intermediate and the gradual source languages have no *"more distinguishing power over fully annotated terms than static contexts do."*

  However, this requirement is quite strong, as the authors recognize that even the original language of Siek and Taha (2006) cannot be considered "good" according to this approach, as the gradual language gains distinguishing power: in the original language (STLC) all programs terminate, but in the gradual version programs may diverge.

2.3.4 *Oh no why is this so slow*

One of the original goals of gradual typing was to use the available type information to optimize programs and make them faster. But as sound gradual typing systems developed, users observed that sometimes their programs, instead of getting faster because of the type information, kept getting slower. Takikawa, Feltey, et al. (2016) discuss the overheads of gradual typing and how prohibitive they can become in some cases, after analyzing all possible combinations of static and dynamic typing in a migratory typing based environment. Many optimizations to reduce these overheads are being studied (Allende

et al., 2014; Bauman et al., 2017; Campora, S. Chen, and Walkingshaw, 2018; Feltey et al., 2018; Moy et al., 2021; Richards, Arteca, and Turcotte, 2017).

I do not particularly focus on the efficiency or overheads of our approach in this dissertation, as my interest is focused on getting some semantic guarantees about the language and its error reporting. I agree with the sentiment of Tobin-Hochstadt, Felleisen, et al. (2017), not only for migratory typing, but for gradual typing as well:

> *"While industrial researchers may trade ideals for practical concerns, especially performance, academic researchers have the moral obligation to strive for them – because nobody else will and society affords them exactly this luxury with generous support. In the context of migratory typing, soundness is the critical ideal."*

2.4 GRADUAL LANGUAGES BY CALCULATION: ABSTRACTING GRADUAL TYPING

I introduce, at a high level, the Abstracting Gradual Typing (or AGT) framework, originally developed by Garcia, Clark, and Tanter (2016). AGT is a general framework for systematically designing a gradually typed language. The AGT framework can be thought of as an algorithm applied to a statically typed language to produce a gradual language.

The standard AGT framework produces a gradual language (a type system plus a runtime) based on the following inputs:

- A statically typed language specification. AGT assumes we begin with a system that already provides at least *some* guarantees: this specification is encoded in a type system plus an operational semantics (a language runtime specification) and a proof of *type safety*.

- A syntax for *gradual types* and a definition of their meaning. In AGT, the meaning of a gradual type (encoded in a *concretization function*) is expressed in terms of a set of static types. This set of static types represents the range of static uncertainty we are willing to accept for a particular gradual type, and encodes the notion that any static type in the meaning could appear in place of a gradual type at run time. Note that this set could be infinite, like in the case of the standard unknown type '?', which means "any possible type could appear here at run time".

- The statically typed language must use some operations and predicates among types, which are the core feature that we expect AGT to make gradual for us. We must make sure that all uses of the operations and predicates that we intend to make gradual have been made explicit in our statically typed language specification before applying the algorithm.

Given these inputs, AGT produces a family of artifacts that constitute a complete gradual language. The language generated can be considered as a reference interpreter, one that can be later optimized. The automatically generated artifacts include:

- A notion of gradual type precision, a binary relation that we can use to compare gradual types.

As we define the meaning of gradual types in terms of sets of static types, we can compare precision directly via set containment among the meanings of two gradual types.

- Gradual versions of the operations and/or predicates of interest. For the automatically generated predicates over gradual types, these definitions encode the following intuition: the gradual predicate holds if the original static predicate holds for at least *some* static type in the meaning of our gradual types.
- A gradual "source" language, which accepts programs with gradual types.
- A gradual type system for the gradual source language. This gradual type system makes use of the gradual versions of the operations of interest. We must bear in mind that, like in the gradually typed lambda calculus example in Section 2.3.2, we lost transitivity when gradualizing the static operations of interest. Thus a language runtime implementation must re-check throughout evaluation that the static invariants are preserved and still hold.
- Each predicate of interest generates an accompanying notion of *evidence*: a data structure containing sufficient run-time information to guarantee that the static assumptions still hold. AGT usually considers as evidence for an n-ary predicate an n-ary tuple of gradual types. This run-time evidence must be at least as precise as the static assumptions made by the type system, but the key contribution of evidence is that the types in the evidence data structure can be made more precise as execution progresses, because every computation step lets the runtime system learn more about the gradual types at hand.
- To refine the evidence data structure as execution progresses, AGT automatically produces an operation to propagate evidence, called *evidence composition*. Evidence composition acts as a transitivity check over the original operations, and is specified as an operation that recovers the most precise new evidence whose meaning contains **all of the static types** in the meanings of two evidence objects which the static operation transitively holds between them. This composition operation is a partial function that, if successful, should increase the precision of run-time information. It can thus be relied upon to guarantee whether static optimistic assumptions hold throughout computation.[47]
- A second language specification, with syntax and a type system. This language, which acts as an intermediate representation for gradually typed programs, syntactically requires evidence to be sprinkled at every point of the program where reduction can happen.
- A translation from well-typed programs in the gradual source language into the intermediate representation, which sprinkles the assumptions that were made by the type checker into the program.
- AGT delivers an automatically generated operational semantics for the intermediate representation. This operational semantics weaves together the original operational

semantics of the static language with evidence propagation over redexes. If evidence propagation fails, evaluation aborts.

- The intermediate language is type safe by construction, assuming the original language is type safe, and the gradual guarantee theorem discussed in Section 2.3.3.3 holds by construction under this same assumption.

Although AGT provides a lot of "free" benefits, the original system has some limitations:

- The original AGT system may break tail recursion, reintroducing a "space leak" for storing casts, which was a problem already solved for some other gradual type systems. We addressed this issue in Bañados Schwerter, Clark, et al. (2021).
- The original AGT framework *did not provide any detailed error reporting*. Languages designed using AGT abort execution when a run-time inconsistency is found, but they produce only an opaque `Error` message that does not identify any program location. We remedy this issue in this dissertation.

Abstracting Gradual Typing has proven useful and has been applied to develop many gradual systems: refinement (liquid) types (Lehmann and Tanter, 2017), union types (Toro and Tanter, 2017), liquid type inference (Vazou, Tanter, and Van Horn, 2018), dependent types (Eremondi, Tanter, and Garcia, 2019), mutable references (Toro and Tanter, 2020), verification of recursive heap data structures (Wise et al., 2020), data structuring (Malewski, Greenberg, and Tanter, 2021), the calculus of inductive constructions (Lennon-Bertrand et al., 2022), system F (Labrada, Toro, and Tanter, 2022), and a probabilistic lambda calculus (Ye, Toro, and Olmedo, 2023).

2.5 A BRIEF HISTORY OF ERROR REPORTING IN GRADUALLY TYPED LANGUAGES

Semantically, a gradual language does not need to provide detailed error reporting. What gradual languages need to have are enough run-time checks to guarantee that the optimistic assumptions made by the type system hold in practice. These run-time checks are performed whenever values move between different levels of type precision.

Some of the gradual languages we discuss in Section 2.3 abort execution when they encounter a run-time type inconsistency, providing no detailed feedback about the cause of the error. We need to abort execution when finding an inconsistency to guarantee certain theoretical properties like type safety, but just aborting with no details makes debugging harder than it needs to be for developers.

Gradual languages that provide feedback to the developer when a program aborts usually draw from the concept of *blame*. Blame, as we discussed in Section 2.2.3, was originally conceived in the context of behavioural contracts (pre- and post-condition invariants). It does not always suffice to report the line where a run-time check fails to identify the source of a failure, even if doing so were a straightforward modification to a semantics that provides no feedback. In the following sections we discuss which ideas from blame in behavioural contracts have been applied to gradual typing.

2.5.1 *Gradual typing welcomes blame with open arms*

Gradual typing researchers drew from the ideas for error reporting and correctness criteria discovered in the domain of higher-order contracts. Type invariants about functions are similar to higher-order contracts, since missing information about functions cannot be checked until they are applied, just like in the case of contracts. Several efforts have been made to use blame in gradually typed languages.

Tobin-Hochstadt and Felleisen (2006) already introduced a notion of blame for migratory typing,[48] but the idea of blame was first applied to Siek-Taha gradual typing by Wadler and Findler (2009). They introduced the *blame calculus*: a variant of the gradual typing intermediate language of Siek and Taha extended to also track code locations. When propagating run-time type information, the runtime semantics must decide which information must be preserved and which can be safely discarded under global assumptions of correctness.

Wadler and Findler (2009) coin a new motto: *"Well-typed programs can't be blamed"*. We discuss issues with the motto in Section 2.5.3, but the key assumption this motto relies upon is that type declarations are correct. Therefore, they claim that hiding type information at run time (as when placing a value into a more dynamic context) should always be considered "safe". An error may then arise only in the process of checking more precise type information.

Siek, Garcia, and Taha (2009) discuss and compare different designs for blame propagation, providing a different name for the same property as the "blame theorem": *soundness of a particular subtyping relation with respect to a semantics*, where a subtyping relation characterizes safe run-time type checks. Many later authors have therefore called the theorem the "Blame-subtyping theorem".

What are the theoretical guarantees of blame? The original blame work provides weak formal connections between error messages and broken static type invariants, as their arguments act only by negation. The gradual typing blame theorem only characterizes those checks that *will never fail*, making no specific positive claim about the connection between the reported error and a particular cast. Later work detected further issues with the original definition of blame safety by Wadler and Findler (2009), as it did not account for open terms. Zalewski et al. (2020) fix this problem by updating the definition of a blame conjecture[49] to be quantified over any possible well-typed context, guaranteeing that errors will not happen for any valid substitution that provides values for free names in a program. This fix does not change the main focus of the theorem: describing safe casts instead of characterizing failures.

Although limited in metatheoretical scope, blame semantics have been considered by some as the gold standard on error reporting, and most work has either focused on optimization of run-time tracking (Garcia, 2013; Siek, Thiemann, and Wadler, 2015; Siek and Wadler, 2010) or used blame as a requisite for designing new *ad hoc* gradual systems with advanced features (Ahmed, Findler, et al., 2011; Vitousek, Swords, and Siek, 2017).

Just like we discussed in the case of contracts in Section 2.2.3, these theoretical claims about blame do not tell us whether blame is suitable to identify what is wrong in a gradually typed program. Lazarek, Greenman, et al. (2021) transferred the work on blame shifting for contracts of Lazarek, King, et al. (2019) to the realm of gradual typing,

checking again *"whether blame assignment adds any value to a gradually typed language, especially for the benefit of the working programmer."* They mutate programs and model an idealized debugging process, followed by a "rational programmer,"[50] to finally conclude that there exist scenarios in every semantics they explore where blame shifting and the rational programmer approach is unsuccessful.

2.5.2 *Complete monitors in gradual typing*

Our discussion of complete monitoring for contracts in Section 2.2.4 also applies to gradual typing. Greenman, Felleisen, and Dimoulas (2019) transfer the notion of complete monitoring from contracts to gradual types in the migratory typing tradition, coining a new motto in the way, that *"Well-monitored types cannot lie"*. The fine print in this slogan is *what* they cannot lie about. Because complete monitoring guarantees check completeness, they cannot lie about *missing checks*. Well-monitored types can definitely lie about whether the location blamed in a failure should have been blamed, as that property is orthogonal to monitoring.

Greenman, Dimoulas, and Felleisen (2023) expand on the definition of complete monitoring for gradual types, providing crucial high-level guidelines on how to uniformly equip (or "lift") a reduction relation with labels to prove complete monitoring: evaluation must gain labels from the evaluation's data flow, except for base values, which drop labels as they cross a boundary which matches their base type. If the value can fulfill the new required invariant, the value assumes the label of the new side of the boundary at which the value now situates itself.[51] On primitive applications, one can drop the labels of the value arguments to the primitive call and the result assumes the label of the context of the call, just like in the case of contracts.

2.5.3 *Issues with Blame*

Although blame seems to have become a part of gradual typing, I identify several semantic issues with the use of blame for error reporting, and discuss some relevant literature for each of the problems I see. Further general arguments from the literature have been discussed in Sections 2.1 and 2.2. I believe these justify searching for alternative approaches to run-time error reporting in gradual typing that do not rely on blame.

- **What is the *right* blame semantics?** After designing gradual versions of type-and-effect systems in previous work (Bañados Schwerter, Garcia, and Tanter, 2014, 2016), I developed blame tracking systems for our gradual type-and-effect system (Bañados Schwerter, 2016). I identified multiple reasonable semantics for which a blame theorem may be presented, but for which a program failure may result in different error locations being reported as the cause of the failure. For contrast, this very issue led the higher-order contract community to repeatedly refine the theorems proven for a blame strategy (starting from the "central lemma" of Blume and McAllester (2006), to blame correctness and complete monitors). I believe this refinement is not over.

- **Well-typed programs *could* be blamed**. From the beginning, blame in gradual typing has grown from the assumption that *types are correct*. This contrasts with the original motivation of gradual typing: a programmer-controlled migration. If the migration is to be programmer-controlled, we can only assume types are always correct if programmers also never make mistakes when writing type annotations. But they do, and do widely. J.-P. Ore, Detweiler, and Elbaum (2021) show that developers annotate types accurately only about half the time.[52] In a user study with statically typed programmers, Lubin and Chasins (2021) identify that programmers can recognize problems with their previously declared types as their understanding of the problem evolves, leading them to change the types. In these cases, the old types were wrong.

  Sometimes code doesn't even have to change for a program to become wrong, as the context may change: Hoeflich, Findler, and Serrano (2022) report that sometimes changes in libraries or API's *"can cause the types to become inaccurate"*, an instance of a broader limitation with the notion of "bug-introducing code changes" already analyzed by Rodríguez-Pérez et al. (2018). If the context changes, previously well-typed programs that relied on contextual assumptions may become incorrect.

  Dimoulas, New, et al. (2016) identify similar issues with the slogan, providing a specific example:

  > *"Wadler and Findler (2009) contains the catchy but simplistic slogan "well-typed programs can't be blamed". This slogan simplifies the situation so much that it is effectively false. Consider the case where a typed module imports a function from an untyped module and asserts that the function has some type. (...) If the typed module now uses the function and the contract fails, (...) The typed module (...) assumed unwisely that the untyped function lives up to a contract about which the function's author knows nothing."*

  In the example by Dimoulas, blame is assigned to the untyped module, when in fact the assertion in the typed module was the incorrect one, as the dynamically typed portion had no obligation to comply with any type restriction.

  Finally, if the type migration is to be programmer-controlled, programmers should be able to choose to under-specify their type declarations. Sometimes they may also over-specify the types and commit to more than they should have. Because both of these kinds of types are not exactly correct, they could cause a failure, and they *could* be to blame.

- **Sometimes one label is not enough.** Another limitation of most blame theorems in gradual typing has been the oversimplification of error reporting to a single location in the program: if a program has multiple errors, the developer must "pull the thread" through local maxima, fixing one local error at a time with no certainty of making progress towards fixing the program globally.

  A similar issue arises when providing blame for session types (Jia, Gommerstadt, and Pfenning, 2016). Although recent efforts aim to justify a debugging strategy using blame that could lead to the source of a failure (Like the "rational developer" approach mentioned in Section 2.5.2) using a single label, they don't always work. I

refer to the discussion in Section 2.2 for more examples from the literature where one failure point is identified as insufficient.

As type systems grow in complexity, sometimes the problem is not only the number of labels, but the idea of a label altogether. Williams, Morris, and Wadler (2018) replace blame locations with a blame data structure that remembers which part of the contract the failure was in. This is needed when dealing with union and intersection types, as failures may not immediately abort execution but instead cause backtracking to check another possibly valid type constraint. Others in a similar domain (Castagna and Lanvin, 2017) decided to disregard "so-called blame" altogether.

The restriction to a single failure location and the assumption of type annotation correctness together also lead blame semantics to drop information that may be relevant, like the places where invariants got weakened during evaluation.

- **The word "blame" assumes guilt.** However, this is not the case in practice: if blamed locations were always at fault, the blame shifting approach would never be necessary. Muehlboeck and Tate (2017) avoid the issue by renaming "blame" to "accountability".

  Muehlboeck and Tate (2017) also argue against delayed blame in general, and for "immediate accountability" instead: as no checks need to wait to be resolved, this approach has the benefit of limited overhead compared to natural sound gradual typing, though it cannot account for higher order checks that must be unavoidably delayed. By restricting the types that can be used gradually to only those that can be immediately checked and by excluding, for example, function types, they completely avoid the overheads of the delayed checks that those types would need.

- **Blame has an ordering bias, like compiler error messages in type-inferencing systems.** S. Chen and Campora III (2019) connect blame tracking with the bias issue in error reporting with type inference we discussed in Section 2.2.1.2, caused by the order in which the constraints are resolved. This bias may result in an error message that may be misleading.

- **The metatheory of blame does not justify error messages, only characterizes safe subprograms.** As mentioned in Section 2.5.1, blame systems provide limited guarantees: the blame-subtyping theorem just says that certain code should never be blamed, but it never formally establishes who *should* be blamed for errors or *why*. In fact, the reported error does not need to be related to the source of the problem in a particular execution. Blame theorems are not a sufficient property to justify a run-time error tracking semantics. I explored some general high-level criteria to evaluate blame strategies that go beyond the blame theorem in the past (Bañados Schwerter, 2016), but all the intrinsic issues we have discussed with blame limit the metatheory we can prove based on a blame semantics.

2.6 experimental evaluation in programming languages

This dissertation combines formal and empirical work. For decades, programming language research had been focused on formal aspects of systems without necessarily addressing the human interaction dimensions of programming systems. In recent years the community has begun considering more user-centred approaches: languages are human computer interfaces, thus we can consider programmers as users of an interface (as proposed by Myers, Ko, et al. (2016) and Myers, Stefik, et al. (2016)). As an example, Coblenz, Kambhatla, et al. (2021) provides a framework for user-centred programming language design, which has been applied to asset ownership in Coblenz, Aldrich, et al. (2020).

After we show from a formal perspective that a provenance tracking semantics provides all necessary information to reproduce an error, we investigate whether provenance provides useful feedback to software developers to fix bugs. Unfortunately we cannot make any claims about the users and their uses of the language from a purely formal perspective, which can only tell us whether a theorem is true or not.[53] To provide some insight and evidence for this question, we embrace the approach proposed by Wrenn and Krishnamurthi (2017):

> *"(...) error messages are a human-computer interaction element, they should be subject to user studies and other forms of evaluation."*

The work on user evaluation of gradual typing is just beginning. Hanenberg et al. (2014) compared users programming in statically typed and dynamically typed languages. Tunnell Wilson et al. (2018) compared evaluation semantics appearing in gradual typing with a survey, and showed that developers expect the type annotations to impact the semantic behaviour of their programs.

Several studies have looked at error reporting in other spaces. There is active research on improving error messages for novice programmers by Marceau, Fisler, and Krishnamurthi (2011) and Prather et al. (2017), as well as on providing example witnesses of static type errors by Seidel, Jhala, and Weimer (2016). Several user studies analyze some of the provenance tools that I discuss in Section 2.2.2.3 (Danas et al., 2017; Hu et al., 2020; Ko, Myers, et al., 2006). We discuss other work on errors and types throughout this chapter, and I hope this dissertation contributes to the human dimension of the study of programming languages.

2.7 NOTES

1. We speak of programming systems here as an intentionally broad term of which a programming language is part, following Jakubovic, Edwards, and Petricek (2023): *"an integrated and complete set of tools sufficient for creating, modifying, and executing programs"*.

2. Becker et al. (2019) also collect a series of complaints from the literature about compiler error messages: *"in the last 50+ years, little positive has been said about compiler error messages. In this time they have been described as inadequate and not understandable (1967), useless (1976), not optimal (1976), inadequate (again, 1983), frustrating (2005), cryptic and confusing (2006), notoriously obscure and terse (2007), undecipherable (2010), intimidating (2012), still very obviously less helpful than they could be (2015), inscrutable (2017), frustrating (again) and a barrier to progress (2018)."*

3. Although software engineering folklore associates later errors with higher costs, recent research has shown that cost increases do not always happen (Menzies et al., 2017).

4. Although we can always trace back the notion of types to logic (Russell, 1903), specifically through the introduction of the simple theory of types by Ramsey (1926) and the simply typed lambda calculus (Church, 1940), in programming languages the distinction gets established with the coming of high-level languages. Perlis (1978) places the notion of type in ALGOL 58 as "analogous to that employed in logic", and Knuth and Trabb Pardo (1980) trace back datatypes to Plankalkül (Zuse, 1948). For contemporary comparison, the TYPE-IN keyword in Grace Hopper's MATH-MATIC (Remington Rand Univac, 1958) is only intended to stop a program and wait for user input. The MATH-MATIC manual also uses the term "unityped", but only to refer to text typed using a UNIVAC-branded input device.

5. Although we usually trace dynamic typing back to LISP (McCarthy, 1960), the first language to be considered dynamically typed, there is little contemporary discussion of types. The original draft by McCarthy (1957) avoids the subject entirely (*"We do not discuss how the programmer indicates what kind of quantity a given symbol represents"*). In a later retrospective, McCarthy (1978) recognizes that, although he used the lambda notation from Church (1941), a book which focuses on the untyped lambda calculus (Church, 1932), he *"didn't understand the rest of his book"*. In the same retrospective McCarthy talks about type optimizations, mentioning that *"Some recent versions of LISP allow distinguishing types, but at the time, this seemed incompatible with other features."*
   The term "dynamic types" can be traced back at the very least to discussions about implementing ALGOL, as Wohlfahrt (1962) distinguishes between "Dynamic type treatment", where *"The logical circuitry or programmes for each arithmetic operation test, at run time, the operands for their types before performing any operation in accordance with the types found"*, and "Static type treatment" where the compiler *"generates different programme structures for arithmetic operations in accordance with the type of operands as found at compiling time"*.

6. Although the complete phrase "well-typed programs cannot get stuck" does not appear in Wright and Felleisen (1994), they explicitly say that *"The critical property we seek is that evaluation of a typable expression cannot get* stuck*"* (p. 59).

7. Harper (2013) refers to Dana Scott as introducing the idea that "untyped" or dynamically typed languages are really "uni-typed", in that one may write a one-type type system against which one can claim type safety for the language.

8. Identifying the failure in this context means "if the program had had annotations, the program would have been rejected by the type checker because of the inconsistency. Because the program would have been rejected, the programmer would not have committed the inconsistency to the repo and thus the bug would not have occurred."

9. This is an underapproximation, and although it sounds low, the paper claims it is an exciting figure for industry people.

10. The idea of software contracts is also connected to Hoare logic (Floyd, 1967; Hoare, 1969).

11. *"The creation relation relates the new symbols in T_{i+1} produced by the rewriting step to the nodes of T_i that matched the symbols in the left-hand side of the rewriting rule (making the rewriting step possible)"*, and the *"residuation relation*

relates every other node in T_{i+1} to the corresponding occurrence of the same node in the T_i" (Field, Ramalingam, and Tip, 1995)

12. Their soundness and minimality theorems are restricted to a subset of all possible rewriting systems, namely "left-linear": Every variable occurring in the left (input) side of a rewrite rule occurs once.

13. From a slice of a "subcontext" of the output they produce a subcontext of the input, such that one may reach a (possibly bigger) subcontext of the output containing the original slice of interest via a subset of the original set of reductions.

14. The authors identified a key intuition, that *"Many type errors are actually inconsistencies between the types of two expressions, and it is often the case that several noncontiguous program locations are involved in a type error"* (Tip and Dinesh, 2001)

15. Or as they call it, their type slices are *"semantically well-founded"*: *"type-checking (the slice) is guaranteed to produce the same error"* (Tip and Dinesh, 2001)

16. The authors claim that *"the accuracy of the locations depends inversely on the degree to which the specified type-checker explicitly traverses syntactic structures such as lists."*, an interesting claim, as our work using Galois slicing seems to indicate the opposite. We do agree with their claim that smaller slices tend to be better: *"Accuracy indicates the quality of the slice obtained. Generally, "small" slices, which contain few program constructs, are desirable because they convey the most insightful information."*

17. *"To choose the correct place to fix a type error, the programmer must find all of the other program points that participate in the error. To find these programs points, the programmer must reconstruct the state of the type inference algorithm at the time it failed, and then run the type inference algorithm* backward*"* (Haack and Wells, 2004). Instead of the squares (□) notation that we use in our slicing formalisms, they use "dots" (...) to identify the parts of a program that are sliced out, which may be a better choice in a computer terminal error message.

18. Readers do not need to be familiar with the details of what a semiring is to understand the intuitions that we present here.

19. The name Galois slicing was assigned *post-facto* to this work in Ricciotti et al. (2017)

20. This program can be desugared into the paper syntax of Transparent ML. We choose to extend Transparent ML to simplify the example, assuming an implementation of `nth` based on pattern matching on the list structure. I chose these function names to be consistent with the Standard ML basis library.

21. This issue is at the core of the proposed "Blame shifting" debugging strategy of Lazarek, King, et al., 2019. A blame report can sometimes point to a code location that is completely correct. We discuss blame shifting in Section 2.2.3.

22. A related problem arises in the context of Object-Oriented contracts. Object-oriented languages usually have inheritance, and inheritance is expected to preserve substitutability. In this case, contracts for method pre- or post-conditions need to also be checked for consistency with other contracts for the same method in the inheritance chain, as discussed by Findler and Felleisen (2001) and Findler, Latendresse, and Felleisen (2001).

23. Blame correctness depends on programs following a "single-ownership" discipline (then called just "ownership"). A runtime semantics is "blame correct" if, whenever evaluation reaches the reduction of a value monitored with a flat contract, the label of the owner of the value is among the labels in the contract context. In other words, *"the "positive" label of the monitor and the ownership label on the value must coincide, and, furthermore, the set of obligations for the flat contract must contain the value"*(Dimoulas, Findler, et al., 2011).

24. *"The pragmatic purpose of blame is to narrow down the code that a programmer needs to examine to locate the bug when the contract system discovers a contract violation. Or so the literature on higher-order contracts claims. In reality, however, there is neither empirical nor theoretical evidence that connects blame with the location of bugs."* (Lazarek, King, et al., 2019)

25. *"If blame points to a component, either the component contains the bug, or if a programmer increases the precision of the contracts between the blamed component and those from which it receives values, then blame shifts to another component"* (Lazarek, King, et al., 2019)

26. *"When blame shifts from one component to another due to increasing the precision of contracts, blame moves* closer *to the bug in the program"* (Lazarek, King, et al., 2019)

27. *"The hypothesis violations we discovered indicate that assuming blame trail and search progress hold as properties is not a generalizable strategy."* (Lazarek, King, et al., 2019)

28. Greenman (2020) presents a clear and detailed explanation of the general design for a complete monitoring semantics. I aim to summarize the gist of his recipe in the rest of this section.

29. The "single" in "single-ownership" requires explanation. Although I would prefer to call it something akin to the "one owner at a time, with changes only through monitors" discipline, this suggested name is not as catchy.

30. Work on complete monitor refers to this instrumentation process as "lifting".

31. The key feature that makes a language *optionally typed* is that types are not allowed to impact the runtime semantics in any way.

32. Unfortunately, some optionally typed languages like TypeScript take a step further and choose to not be type safe, so programmers in TypeScript cannot rely on any type declarations, ever.

33. `https://www.dartlang.org`

34. `http://www.typescriptlang.org`

35. `http://hacklang.org`

36. *"A gradual type system accepts that it does not know certain types and inserts [run-time] casts"* (Siek and Taha, 2006)

37. As Tobin-Hochstadt, Felleisen, et al. (2017) describe the difference between these two approaches, *"the purpose of migratory typing is to support the migration of code from an untyped setting to a typed one, while preserving the ability to run any mixed-typed software system with the same guarantees as the fully untyped or fully typed ones. Clearly gradual typing can be used for migrating code in a sound manner, but it is equally well suited for annotating extremely small fragments with types for documentation of logical invariants or for exploratory coding in the context of a fully-typed system."* They also mention that *"outside developers report that modules are too large for a migration effort. People would much prefer to add the type invariants of just a key algorithm to a module when they have to revisit the code. This desire suggests a preference for an approach based on gradual typing rather than migratory typing."* Although the intention of introducing modules as a migration boundary was to minimize costly boundary crossings, this is not necessarily the case when programs interact in ways that generate plentiful boundary crossings between modules anyways. The problem of boundary crossing costs is orthogonal and remains an issue no matter the size of the migration boundary, see Section 2.3.4.

38. `https://github.com/racket/typed-racket`

39. The language is a gradual version of the Simply Typed Lambda Calculus (STLC), originally developed by Church (1940). The STLC language is a favourite template for new designs in the PL community.

40. A type-and-effect system, in the style of Marino and Millstein (2009), is akin to checked exceptions: Parts of the code can specify what side effects are allowed (as with the "throws" clause in Java). In progressive types, the side effects handled by the type system are run-time errors: Annotations can be used to determine what failures are rejected statically and which are delayed until run time.

41. Although the original language with concrete types also had "like types", these two are not the same. Richards, Nardelli, and Vitek (2015) recognize that the like types introduced in Wrigstad et al. (2010) are actually optional types.

42. Though I discuss the blame-subtyping theorem in more detail in Section 2.5.1, I disagree with blame-subtyping being a fundamental property of gradual typing, and I specify some of the issues I see in Section 2.5.3.

43. For some highlights, Ahmed, Jamner, et al. (2017) proved parametricity, leaving the gradual guarantee as future work. Toro, Labrada, and Tanter (2019) identified the gradual guarantee to be incompatible with parametricity, then New, Jamner, and Ahmed (2020) and Labrada, Toro, Tanter, and Devriese (2022) introduce constraints to reconcile them (New, Jamner, and Ahmed (2020) identifies type-directed computation as the problem instead of incompatibility between graduality and parametricity). The incompatibilities are refined in Labrada, Toro, and Tanter (2022).

44. The contrapositive theorem is usually made more precise, connecting the blame error messages between the programs. However, because I am not immediately interested in blame, I focus on the main intention of the theorem here.

45. By deterministic we mean that a program always evaluates to the same value. By type safe we mean that evaluation is defined for all well-typed terms. The two are key to reason contrapositively here. The gradual guarantee provides contrapositive implications whose premises refer to when a program "does not diverge" and "does not evaluate to a value", and determinism and type safety bridge the gap for us to claim properties about "evaluating to an error".

46. Readers unfamiliar with compiler theory may be surprised that this definition of observational equivalence does not inspect the resulting values when programs terminate. This is a standard sleight of hand used in the literature, as when the language is sufficiently expressive, one can always engineer a context akin to "if the values are not equal, diverge for one and not for the other". Therefore, stating the theorem "for every valid context" includes a context that only terminates when values are equal, and proving "equivalence of termination", when contexts are expressive enough, can subsume "equivalence of terminating with the same output". Many languages that we build models of are expressive enough.

47. The monotonicity criterion is not guaranteed in the original work, but we refine the framework in this dissertation to constrain its inputs in a way that guarantees this property.

48. Their semantics reports a blame location, and the theorem they provide reads *"typed modules can't go wrong, and all run-time errors originate in untyped modules".*

49. The work provides no theorem as they provide no proof.

50. *"Like homo economicus, the rational programmer approximates the behavior of a software developer who reduces time spent on a task by exploiting the available information [...]: the error message and the state of the program. Hence, the most rational procedure is to use the former to improve the latter. Specifically, the rational programmer translates the Wadler-Findler slogan into a debugging method."* (Lazarek, Greenman, et al., 2021). Although I am not an economist, I join many economists in disagreeing with the assumptions of "homo economicus" as a model. It is also not clear yet whether evidence of rational programmers exists in practice.

51. According to Greenman, Felleisen, and Dimoulas (2019) complete monitors are designed to ensure transfer of responsibility for the fulfillment of a type invariant (considered as an obligation). *"If the obligations can be discharged, the transfer is complete and the receiving component takes on full responsibility—or ownership. If not, both components co-own."*

52. 51% $\pm$ 8.5%, with a Mechanical Turk experiment focusing on first-order physical measuring unit types. Previous experience of participants has little impact on accuracy, and making several related type annotations does not increase accuracy significantly but increases speed. The main clues for type selection are identifier names and reasoning over code operations.

53. *"One can't proceed from the informal to the formal by formal means"* (Perlis, 1982)

I always assumed gradual types were to help those poor schmucks using untyped languages to migrate to typed languages. I now realise that I am one of the poor schmucks.

— Wadler (2015)

3 FROM AGT TO PRECISE AGT

The original AGT framework allows language designers to use imprecise abstractions for their design of gradual types. With imprecise abstractions, the AGT recipe produces a semantics that is unable to remember throughout evaluation all the assumptions that accrue as a program runs. To see the subtleties of the problem, we must first be familiar with the detailed workings of the AGT framework.

I first present the details of the original AGT recipe and how to use it concretely by applying it to an introductory example language: the simply typed lambda calculus (STLC). The result is a gradual language we call the Abstracting Gradually Typed Lambda Calculus (AGTLC). This language is very similar to, but not the same as, the Gradually Typed Lambda Calculus originally introduced by Siek and Taha (2006). After introducing the recipe in detail, we revisit the language design introduced by Garcia, Clark, and Tanter (2016), $GTFL_{\lesssim}$, identify how that language exposes limitations with the original recipe, and introduce restrictions to the abstractions. These restrictions can be used to avoid the problem.

3.0.1 *Peeking at the problem with plain AGT*

AGT provides a specification for a type safe runtime semantics that performs type consistency checks, but sometimes the AGT-based run-time checking regime requires manual tuning to ensure that all desired and expected *modular type-based semantic invariants* are properly enforced. AGT promises that the gradual runtime will enforce static invariants, but in discussing $GTFL_{\lesssim}$ in Section 3.2, we observe that the original AGT approach does not always live up to its promises, requiring extra work and designer insight to achieve some of the semantic goals for a gradually typed language.

This chapter refines the recipe to avoid *ad hoc* tuning by introducing guidelines that impose stricter constraints on the inputs to the AGT recipe. The constraint, forward completeness, guarantees the enforcement of invariants accrued as programs run. In refining the recipe, I introduce a new abstraction for gradual record types that I call Bounded Rows and Records (or BRR). BRR enforces these semantic invariants in addition to the criteria of Siek, Vitousek, et al. (2015) that AGT enforces by design.

The fact that AGT requires extra tuning has been observed in prior work. When gradualizing sophisticated typing disciplines like information flow security (Toro, Garcia, and Tanter, 2018) or parametricity (e.g., Ahmed, Jamner, et al. (2017), New, Jamner, and Ahmed (2020), and Toro, Labrada, and Tanter (2019)), applying AGT naïvely does

not automatically enforce the modular invariants that a programmer desires from the intended static type discipline. Formally speaking, information-flow and parametricity are *hyperproperties*: relationships among **multiple runs** of a program. Because AGT was not designed to enforce hyperproperties, it may be expected that these disciplines may break, but we are first to show that these invariants may also break in simpler languages.

Applying the AGT recipe needs extra tuning also for the original AGT gradual language introduced by Garcia, Clark, and Tanter (2016), where intended invariants can also break (we introduce that language, GTFL$_{\lesssim}$, in Section 3.2). The type discipline of GTFL$_{\lesssim}$—record subtyping—is comparatively simple and well-understood. The AGT-based gradual language is type-safe, but it does not preserve all the type boundaries present in the original language.

The following scenario demonstrates the problem. Consider this program fragment:

$$\mathsf{let}\ \mathsf{q} : [\mathsf{x} : \mathsf{Int}] = [\mathsf{x} = 5, \mathsf{y} = \mathsf{true}]\ \mathsf{in}\ \langle \mathbf{body} \rangle$$

According to standard subtype-based reasoning, the body cannot access the y field of the record because the type annotation for q exposes only one field: x. Such modular reasoning is a hallmark of static typing, and there is some evidence that programmers who mix static and dynamic typing want to reason about their code using static types where possible (Tunnell Wilson et al., 2018).

Unfortunately, the following completed GTFL$_{\lesssim}$ program compiles and runs successfully to true, even though we would expect the program to compile and produce a run-time type error for breaking the encapsulation that the declared types should enforce:

$$\begin{array}{l} \mathsf{let}\ \mathsf{q} : [\mathsf{x} : \mathsf{Int}] = [\mathsf{x} = 5, \mathsf{y} = \mathsf{true}]\ \mathsf{in} \\ \quad \mathsf{let}\ \mathsf{z} : ? = \mathsf{q}\ \mathsf{in} \\ \quad\quad \mathsf{z.y} \end{array}$$

The identifier z is ascribed to the *unknown type* ?, which marks absent type information and enables run-time checking.[1] The problem with this program is that we expect the type annotation for q to impose an encapsulation boundary, hiding field y throughout the body of the outermost let. But the ascription to type ? combined with the design of gradual types in GTFL$_{\lesssim}$ breaks that encapsulation boundary: the abstraction is imprecise and loses information.

How does this problem manifest in this example program? As discussed in the introduction, a gradual type in AGT is an abstract representation of a set of static types, in this case, related via subtyping . When binding z at type ?, we can actually see the type of z as representing all possible upcasts for q, and in this case there are only two: either we can see q as the declared $[\mathsf{x} : \mathsf{Int}]$ or hide its only field at type $[]$. The best that GTFL$_{\lesssim}$ can do to represent these two cases together is to combine them in a gradual type that represents *any* valid record type, not just these two, leaking our desired encapsulation.

A gradual type system does not see a problem when checking the body of the innermost let expression. Because the type of z is ?, the type system can only see that *any static type with the correct field would work* when trying to access the field y of the z value by using the record projection operation (the dot). This is what we want in a gradual language: the

problem only materializes when running this program. When combining the information available for z at run time with the information available for q, the runtime forgets that the field y was supposed to be inaccessible. The type restrictions declared for q are forgotten, because the design of gradual types in GTFL$_{\lesssim}$ cannot restrict the uncertainty among the abstracted static types to *only* the field x.

In essence, we would like the language to remember that q was only to be seen as $[x : \mathsf{Int}]$ whenever we cast it to the unknown type and then back to a record type. Instead, the original design regains access to any part of the record when we cast q to the unknown type and then back to a record type, including those fields that were to be encapsulated away by ascribing q to the type $[x : \mathsf{Int}]$. The type constraint that the programmer specified for q is forgotten because the run-time abstraction is not as precise as it needs to be for the subtyping relation it aims to model. In other words, the language satisfies standard type safety, as AGT does by construction, but its run-time treatment of type abstractions does not strictly enforce the type abstraction properties one expects from subtyping.

Previous AGT-based systems like Toro, Garcia, and Tanter (2018) and Toro and Tanter (2020) have had to refine their run-time abstractions guided by designer's intuition. We develop a principled approach for tuning the run-time checks in an AGT-based language, and show that it suffices to address the challenge of enforcing abstraction.

The rest of this chapter is organized as follows:

- We introduce AGT by first applying it to a simple language: the simply typed lambda calculus. Doing so, we obtain the AGTLC language in Section 3.1.

- We discuss the gradual language with record subtyping introduced by Garcia, Clark, and Tanter (2016) and its problems in Section 3.2.

- We present in Section 3.2.5 a simple setting where AGT-designed semantics do not automatically ensure all desired type-based reasoning principles. Subtyping is a standard unary type property that still requires care to enforce when gradual types are sufficiently rich. This problem had been previously observed only on sophisticated typing disciplines that enforce properties that are fundamentally about multiple runs of a program (i.e., hyperproperties).

- We revisit the foundations of AGT's dynamic checks and refine their conception in Section 3.3. Garcia, Clark, and Tanter (2016) conceive of them via *post hoc* manipulation of abstractions for tuples of types: we redefine them as a direct abstract interpretation of *the subtyping relation itself*. This reframing is subtle and equivalent to the original, but by doing so techniques from the abstract interpretation literature become immediately applicable to the type-based reasoning problems discussed in Section 3.2.

- We explore the precision issue in the context of Garcia, Clark, and Tanter (2016)'s GTFL$_{\lesssim}$ language in Section 3.4. We introduce *bounded records and rows*, a new abstraction for run-time evidence that retains precise information about gradual types and how they interact at run time. Bounded rows uncover a subtle interplay between static record subtyping and *gradual row types* that went unnoticed in previous work.

- We devise sufficient conditions for precise run-time monitoring in a general form in Section 3.5. General conditions are useful as AGT is a general-purpose framework for designing languages Precise run-time monitoring can be achieved, while satisfying the formal criteria of Siek, Vitousek, et al. (2015), by ensuring that the dynamic monitoring semantics is *forward complete*, a concept from abstract interpretation (Giacobazzi and Quintarelli, 2001). With forward completeness, reasoning about the dynamic semantics is equivalent to reasoning up to precision ($\sqsubseteq$) about static subtyping.

A considerable portion of the work in this chapter has already been published in Bañados Schwerter, Clark, et al. (2021). That paper also discusses space-efficiency for a precise AGT language. I do not discuss space efficiency in this dissertation as it does not directly relate to my thesis, but I direct readers interested in the space efficiency dimension of this problem to the original publication.

3.1 AGTLC: APPLYING AGT TO THE SIMPLY TYPED LAMBDA CALCULUS

I show by example how AGT works. I apply AGT to a standard language, the simply typed lambda calculus (or STLC), to obtain a gradual language from it.

Why STLC? First, it is a simple but expressive language that exhibits difficulties arising in many more advanced languages, particularly dealing with delayed computation and naming issues (in this case, via substitution). Second, it was the first language made gradual by Siek and Taha (2006). Third, it is a standard language familiar to most programming language designers and is featured in most textbooks (like Benjamin C Pierce et al. (2019) and Benjamin C. Pierce (2002)). Finally, even for this simple language, AGT already shows semantic differences with *ad hoc* designed gradualizations like the ones explored and compared by Greenman, Dimoulas, and Felleisen (2023).

Applying the AGT recipe consists of a series of steps:

1. Rewrite the static language in an AGT-friendly style: outputs expected from premises should be made opaque, and every reference to a type predicate or function must be explicit in the static type system. Typing rules cannot implicitly depend on pattern matching premise outputs, any constraints of this sort (like "the first expression in a function application should have a function type") should be made in terms of type predicates and functions so that they can later be turned into their corresponding gradual versions.

2. Collect the "design ingredients" for the AGT recipe: that is, all the portions that a language designer needs to actually design and provide to the AGT framework as inputs. The main ingredient here is the design of gradual types and their meaning, and selecting the type predicates and functions that will be made gradual instead of statically checked.

3. Obtain from the AGT recipe a specification of gradual type predicates and functions based on the previous design ingredients.

4. Obtain from the AGT recipe a gradual surface language and a type checker for it, based on minor recipe-guided alterations to the original static type checker.

5. Obtain a runtime language that is both type safe and propagates run-time type information. The runtime uses this type information to identify inconsistencies that the type checker could not immediately see in the original program.

6. Obtain a type-directed translation from the surface language into the runtime language to provide a dynamic meaning for programs.

The rest of this section discusses each of these steps in detail.

3.1.1 *Preparing STLC for AGT*

In Figure 3.1, we present the simply typed lambda calculus ready for AGT. The syntax is presented in standard Pānini-Backus form.[2] The language needs a base type and its corresponding values, which in this case are numbers. The key feature this language has is functions: both anonymous functions (lambdas) and a function application construct to use them. Functions bind identifier names and encode delayed computations. The language also has type ascriptions, which can be used to declare the type a sub-expression may have. The runtime semantics is defined in terms of notions of reduction that can be applied under evaluation contexts. There are two basic ways to reduce expressions in a program: nested ascriptions and beta-reduction (that is, function application).

To enable the use of the AGT recipe, the type system of the static language of interest takes a particular form. Language designers usually rely on pattern matching in inference rules to encode extra constraints in an inference rule: For example, the usual typing rule for function application in STLC is presented as follows:

$$\text{App}\frac{\Gamma \vdash t_1 : T_0 \to T_3 \qquad \Gamma \vdash t_2 : T_0}{\Gamma \vdash t_1\ t_2 : T_3}$$

This rule design encodes implicit constraints: first that the type of t_1 should have the shape of a function, and that the type of t_2 should be the same as the domain of the type of t_1. Now, for AGT to work, all of these constraints must be made explicit separately: The output types of premises remain opaque as arbitrary outputs, and the constraints are expressed separately, as in the [App] rule in Figure 3.1.

By changing the type system in this way, we have unfortunately lost a property that STLC had: the set of inference rules defining the type system was *well moded*,that is, we were able to build a search procedure for type checking by separating some parts of the judgment as inputs (the context and the expression) and others as outputs (the type had by the expression). With the constraint-based design as presented in Figure 3.1, rules are not well moded: we cannot resolve the outputs from the inputs of the judgment alone.[3] To ensure that the new rule for [App] is well moded, the equality predicates in premises have to be checked, so both of the types act as inputs.[4] A type checker would need to come up with some types T_0 and T_3 out of somewhere, as these are not inputs from the typing judgment.

We can rewrite the type system in a *different* constraint-based way, such that the type system preserves its well moded structure, while also making all typing constraints explicit. We use instead helper type functions as follows:

Syntax

$$x \in \textsc{Var}, \quad n \in \mathbb{Z}, \quad \Gamma \in \textsc{Env} = \textsc{Var} \xrightarrow{\text{fin}} \textsc{Type}$$

$$\begin{array}{rcll}
T \in \textsc{Type} & ::= & \mathsf{Int} \mid T \to T & \text{(static types)} \\
t \in \textsc{Term} & ::= & n \mid x \mid \lambda\,(x : T)\,.\,t \mid t\ t \mid t :: T & \text{(terms)} \\
u \in \textsc{RawValue} & ::= & n \mid \lambda\,(x : T)\,.\,t & \text{(raw values)} \\
v \in \textsc{Value} & ::= & u \mid (u :: T) & \text{(values)} \\
E \in \textsc{ECtxt} & ::= & \Box \mid E[F[\Box]] & \text{(evaluation contexts)} \\
F \in \textsc{EvFrame} & ::= & \Box\ t \mid v\ \Box \mid \Box :: T & \text{(evaluation frames)}
\end{array}$$

$\boxed{\Gamma \vdash t : T}$ **Static Typing**

$$\text{Int}\frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \text{Id}\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{Lam}\frac{[x \mapsto T_1]\Gamma \vdash t : T_2}{\Gamma \vdash \lambda\,(x : T_1)\,.\,t : T_1 \to T_2}$$

$$\text{Ann}\frac{\Gamma \vdash t : T_1 \qquad T_1 = T_2}{\Gamma \vdash t :: T_2 : T_2}$$

$$\text{App}\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad T_1 = T_0 \to T_3 \qquad T_2 = T_0}{\Gamma \vdash t_1\ t_2 : T_3}$$

$\boxed{t \rightsquigarrow t}$ **Notions of Reduction**

$$(\lambda\,(x : T)\,.\,e)\ v \rightsquigarrow [v/x]\,e \qquad (u :: T_1) :: T_2 \rightsquigarrow (u :: T_2)$$
$$((\lambda\,(x : T_1)\,.\,e) :: T_2)\ v \rightsquigarrow [v/x]\,e$$

$\boxed{t \longrightarrow t}$ **Contextual Reduction**

$$\text{Structural}\frac{e \rightsquigarrow e'}{E[e] \longrightarrow E[e']}$$

Figure 3.1: The Simply Typed Lambda Calculus with annotations, in a style ready to apply the AGT recipe

$$\text{App}\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad T_2 = dom(T_1)}{\Gamma \vdash t_1\ t_2 : cod(T_1)}$$

Using this rule, a type checker does not need to come up with any new types for the constraints, they all arise from the outputs of recursively type-checking sub-parts. For reasons that will become clear as we apply the recipe, we keep *both type systems*, and a translation for typing derivations from the well moded system into the specification one. When applying the recipe, the well moded one forms the base for a type checker for the source language, and the specification one forms the base for the runtime system: At that point the lack of a search procedure to build derivations is not a problem *because we already have a well moded type system for the source gradual language and a way to translate from a typing derivation in the well moded system into another typing derivation in the specification type system*.

The basic property that the STLC has is type safety:

Theorem 1 (Type Safety)**.** *The evaluation of any closed well-typed program never gets stuck. That is, any closed well-typed program is either a value, it reduces to a value, or it diverges.*

The proof of this theorem is standard, and we show some of its details when needed as we apply the recipe.

3.1.2 *Collecting the ingredients for the AGT recipe*

The first step in making our AGT gradual language is to design what the gradual types are and what they mean. Every static type should also be a gradual type, but the question is what other types to include.

For this language we choose to define gradual types following the structure of the static types, adding an extra type representing "unknown type information", just like in the original work by Siek and Taha (2006). Note that we don't *have to* write gradual types this way: doing so is a design choice that helps us produce a language comparable to GTLC:

$$S \in \text{GType} \quad ::= \quad ? \mid \mathsf{Int} \mid S \to S \quad \text{(gradual types)}$$

In AGT, the meaning of gradual types is defined in terms of *sets of static types*, so that a gradual type represents some possible *concrete* static types. In this way, we endow gradual types with a semantics: each gradual type S is given an interpretation as a non-empty set of static types from the static language, denoted $\mathcal{P}^+(\text{Type})$.

A *concretization* function assigns meaning to gradual types and is usually expected to act as an extension of the original system.

Definition 6 (Concretization for AGTLC)**.** *We define the meaning of gradual types with the following function:*

$$\begin{array}{rcl} \gamma & : & \textsc{GType} \to \mathcal{P}^{+}(\textsc{Type}) \\ \gamma(\mathsf{Int}) & = & \{\, \mathsf{Int} \,\} \\ \gamma(S_1 \to S_2) & = & \{\, T_1 \to T_2, \textit{ where } T_1 \in \gamma(S_1) \textit{ and } T_2 \in \gamma(S_2) \,\} \\ \gamma(?) & = & \textsc{Type} \end{array}$$

This design follows a recipe that is standard in the design of AGT languages:

- Atomic gradual types represent singleton sets of atomic static types. In this way, the properties of atomic-type values do not get diluted when defining the gradual type system.
- Compound gradual type constructors preserve the structure of a mirroring compound static type constructor but allow flexibility on their subparts. This way, a gradual structural type can describe partial information about the type while still distinguishing its top-level structure. In our case there are only function types, but the same recipe applies to any other structural types, like tuples and the records we see in Section 3.2.
- We introduce extra types with non-singleton meaning. Here we only introduce ?, but we see further gradual type designs later with a more restricted structure (gradual and bounded rows) in Sections 3.2 and 3.4. These extra types are chosen by the language designer, but often there is at least a way to declare an unknown type representing "any static type", as the least possible information available statically, which is the meaning of the gradual type ? here.

As is now standard for gradual typing, the concept of "imprecise type information" is formalized using the *precision* judgment $S_1 \sqsubseteq S_2$, which says that S_1 is less imprecise than S_2.[5] When starting from scratch, concretization is typically defined by recursion on the structure of gradual types (Garcia, Clark, and Tanter, 2016), and then the precision relation is determined by the following correctness criterion:

Proposition 2 (Garcia, Clark, and Tanter, 2016)**.** *A gradual type is less imprecise than another if and only if their concretization images are related by set containment ($S_1 \sqsubseteq S_2$ if and only if $\gamma(S_1) \subseteq \gamma(S_1)$).*

This proposition takes the concretization function γ as the ground truth, and defines precision in terms of set containment among images of concretization. For AGTLC, this means that the definition of precision depends on Definition 6. Another approach to define the meaning of gradual types would be to define a notion of precision directly as a relation between gradual types, and then take that definition of *precision* as the ground truth to define concretization in terms of precision. Both approaches are equivalent, though a structurally recursive definition of concretization is critical to calculating inductive definitions of relations on gradual types and recursive definitions of functions, which our type system requires.

When precision is taken as the ground truth, concretization is then defined as

$$\gamma : \textsc{GType} \to \mathcal{P}^{+}(\textsc{Type})$$
$$\gamma(S) = \{\, T \in \textsc{Type} \mid T \sqsubseteq S \,\}\,.$$

Whichever approach we take, we design the precision partial order by considering static types as the *least* elements of this partial order, while the greatest element is the unknown type ?. Thus, a gradual type denotes every static type that it represents. Since every static type is a (minimal) gradual type, this implies that $\gamma(T) = \{\, T \,\}$.

3.1.3 *First result from the AGT recipe: consistent type predicates*

We can use the concretization function to obtain gradual versions of the static type predicates that were used to define the original static type system. This is the first result from the AGT recipe.

AGT allows us to expand static type relations to relations on gradual types by relying on the concretization function to define the *consistent* version of the original predicates: A consistent predicate holds when the original static predicate holds for at least some static type(s) in the meaning of the gradual types at hand.

STLC depends only on one static predicate, *type equality*. Thus we need to obtain only one gradual predicate for AGTLC, *consistent type equality*.

Definition 7 (Consistent Type Equality in AGTLC)**.** *We say that two gradual types are* consistently equal *if there are some static types in their corresponding concretizations that are equal.*

Formally, $S_1 \sim S_2$ if there exists some $T_1 \in \gamma(S_1)$ and $T_2 \in \gamma(S_2)$ such that $T_1 = T_2$.

This specification provides a definition of consistency equivalent to the one introduced by Siek and Taha (2006), and which can then be expressed inductively.

3.1.4 *Second result from the AGT recipe: Consistent type functions*

The STLC static type system also relies on two partial functions over static types, *dom* and *cod*. As with predicates, we define functions for gradual types in terms of the static type functions of the original language. We make a gradual version of the functions by applying those static type functions to each static type in the meaning of the gradual types, but the tools we have so far do not help us to go from that resulting set of static types to some resulting gradual type. For this we use an *abstraction* function.

An abstraction function obtains the most precise gradual type whose meaning contains all of the static types we intend to abstract. We distill these constraints into two properties that an abstraction and a concretization function have with respect to each other:

Proposition 3 (Garcia, Clark, and Tanter, 2016)**.**

1. *Soundness: The gradual type $\alpha(\mathcal{C})$ summarizes the set of static types $\mathcal{C}$. Formally, a round-trip starting from an arbitrary set of static types can only add elements to the set. By round-trip we mean obtaining a new set of static types from a set $\mathcal{C}$ by applying first abstraction and then the concretization function. ($\mathcal{C} \subseteq \gamma(\alpha(\mathcal{C}))$)*

2. *Optimality: No other gradual type summarizes $\mathcal{C}$ more precisely than $\alpha(\mathcal{C})$. Formally, the round-trip must preserve the containment order for all concretization image supersets (it cannot add too-many elements), or if $\mathcal{C} \subseteq \gamma(S)$ then $\gamma(\alpha(\mathcal{C})) \subseteq \gamma(S)$.*

These two properties are equivalent to requiring that the abstraction and concretization functions form a *Galois connection* (P. Cousot and R. Cousot, 1977; O. Ore, 1944), under certain mathematical constraints.[6] Our choices of functions satisfy the stronger property that $\alpha(\gamma(S)) = S$. This makes them a *Galois insertion*.

The following specification describes an abstraction function. In most cases, the Galois connection theory also guarantees that for any concretization function, the abstraction function is unique.[7]

$$\alpha : \mathcal{P}^{+}(\textsc{Type}) \to \textsc{GType}$$
$$\alpha(\mathcal{C}) = \bigsqcap \{\, S \in \textsc{GType} \mid \mathcal{C} \subseteq \gamma(S) \,\}$$

In this definition, $\bigsqcap$ denotes the greatest lower-bound with respect to precision $\sqsubseteq$. Abstraction $\alpha(\mathcal{C})$ yields the least imprecise gradual type that summarizes a given set of static types. In order for α to be well-defined, we must ensure that the right-hand side of the above equation is well-defined for any collection $\mathcal{C}$ to be considered: in particular there must exist an appropriate greatest lower-bound. In general, a gradual type system does not need to account for every arbitrary subset of static types $\mathcal{C} \in \mathcal{P}^{+}(\textsc{Type})$ as part of concretization and abstraction. In fact, in Section 3.4.3 it becomes critical to restrict the space of sets of types (with no loss of expressiveness) to ensure that α is well-defined.[8] If abstraction α is well-defined, then it is uniquely determined by concretization γ. To facilitate the calculation of algorithmic presentations of operators and relations on gradual types, abstraction is typically presented as a recursive function definition (Garcia, Clark, and Tanter, 2016).

This abstraction function completes our recipe for the definition of gradual functions: We concretize, apply the static function pointwise, and then abstract back the set of static type results to a gradual type that represents them. We then obtain a gradual function for gradual types from a function for static types.

3.1.4.1 *Gradual helper functions for AGTLC*

We use this recipe to define gradual versions of the helper functions that we used in the static type system for STLC. For example, consider the codomain of a function type, rendered as a partial function *cod* on static types:

$$\mathit{cod} : \textsc{Type} \rightharpoonup \textsc{Type}$$
$$\mathit{cod}(T_1 \to T_2) = T_2$$
$$\mathit{cod}(T) \text{ undefined otherwise}$$

We can idiomatically *lift cod* to a partial function cod^+ on non-empty sets of types.

$$
\begin{aligned}
&cod^+ : \mathcal{P}^+(\textsc{Type}) \rightharpoonup \mathcal{P}^+(\textsc{Type}) \\
&cod^+(\mathcal{C}) = \mathcal{C}' \ \text{ if } \mathcal{C}' \neq \emptyset \\
&cod^+(\mathcal{C}) \text{ undefined if } \mathcal{C}' = \emptyset \\
&\text{where} \quad \mathcal{C}' = \{\, T' \in \textsc{Type} \mid T \in \mathcal{C} \text{ and } cod(T) = T' \,\}
\end{aligned}
$$

The cod^+ function applies *cod* pointwise to a set of types and produces the resulting set of codomains, so long as there are some codomains: if not, then the function is undefined. Using this lifted function, we can define the $\widetilde{cod}$ function as follows.

$$\widetilde{cod}(S) = \alpha(cod^+(\gamma(S))).$$

Analogous constructions can be used to first define and then calculate algorithms for all of the operations on gradual types we need. The definitions directly imply their own correctness criteria, and the corresponding algorithms require tedious but straightforward calculations. In the case of AGTLC, we require only two such functions: *dom* and *cod*.

When following the AGT recipe, these functions do not need new design insight. They are defined in terms of elements that we have already designed. These include the concretization function, the static version of the functions, and the abstraction function that is determined by the concretization function.

3.1.4.2 *Correctness criteria for helper functions*

Why are these functions we calculated the right ones? Consider, for instance, how $\widetilde{cod}(S_1)$ extends the idea of "the codomain of a function type". For function types $S_1 \to S_2$, its behaviour is as expected, but for the unknown type ?, the codomain is completely unknown because the operator's type is completely unknown: the operator may have a function type, but we do not know for sure. We formalize this behaviour as a correctness criterion.

Definition 8 (Candidate Codomain)**.**

1. *A* plausible function type *is a gradual type S that is connected to a function type via precision. Formally, $T_1 \to T_2 \sqsubseteq S$ for some $T_1, T_2 \in \textsc{Type}$;*

2. *A gradual type S' is a* candidate codomain *of a plausible function type S if for all function types that are less imprecise than S, their codomain is also less imprecise than S'. Formally, $T_1 \to T_2 \sqsubseteq S$ implies $T_2 \sqsubseteq S'$, for all $T_1, T_2 \in \textsc{Type}$.*

Proposition 4 (Codomain Correctness)**.** *A gradual type S has a candidate codomain if and only if it has a candidate codomain that is a least upper bound (with respect to $\sqsubseteq$) among all candidates, denoted $\widetilde{cod}(S)$.*

Proposition 4 implicitly defines $\widetilde{cod}$ as a partial function on GTYPE, which is defined for exactly the plausible function types. The notion of "candidate codomain" can be interpreted as a *soundness* property of $\widetilde{cod}$: it broadly approximates the idea of "codomain" even in the face of imprecision. The proposition can then be also interpreted as an

optimality property: there is a "best" candidate codomain, if any, so $\widetilde{cod}$ never loses precision needlessly. Analogous correctness criteria apply to the other type operators, completely characterizing the gradual versions in terms of their static counterparts.

The specifications for gradual functions we obtained can also be implemented directly and in an equivalent form that does not depend on calls to abstraction and concretization:

Helper Functions

$$
\begin{array}{ll}
\widetilde{dom} : \textsc{GType} \rightharpoonup \textsc{GType} & \widetilde{cod} : \textsc{GType} \rightharpoonup \textsc{GType} \\
\widetilde{dom}(S_1 \to S_2) = S_1 & \widetilde{cod}(S_1 \to S_2) = S_2 \\
\widetilde{dom}(?) = ? & \widetilde{cod}(?) = ? \\
\widetilde{dom}(S) \text{ undefined otherwise} & \widetilde{cod}(S) \text{ undefined otherwise}
\end{array}
$$

3.1.5 *Obtaining a gradual surface language*

We need no further inputs to define a gradual surface language and its type system. We take the syntax of the original static language and change any instances of static types to gradual types. This process produces the following new syntax for our gradual surface language.

Syntax

$$\Gamma \in \textsc{Env} = \textsc{Var} \overset{\text{fin}}{\rightharpoonup} \textsc{GType}$$

$$
\begin{array}{rcll}
t \in \textsc{Term} & ::= & n \mid x \mid \lambda\,(x : S)\,.\,t \mid t\ t \mid t :: S & \text{(terms)} \\
v \in \textsc{Value} & ::= & n \mid \lambda\,(x : S)\,.\,t & \text{(values)} \\
F \in \textsc{EvFrame} & ::= & \square\ t \mid v\ \square \mid \square :: S & \text{(evaluation frames)}
\end{array}
$$

We then introduce a gradual type system for this surface language by just taking the original STLC static type system and replacing every reference to static type predicates and functions with their gradual versions, yielding the type system of Figure 3.2.

The rules for the typing judgment $\Gamma \vdash t : S$ (read "term t has gradual type S under context Γ") are structured in the style of Garcia and Cimini (2015): each typing judgment in the premise of a rule has an arbitrary type metavariable S_i, but these premise types are constrained by side conditions. The result type in the conclusion of a rule is either a particular type or a (partial) function of its premise types. For example, the two premises of the (Sapp) rule have types S_1 and S_2 respectively; these premise types are constrained by the *consistent equality* side condition $S_2 \sim \widetilde{dom}(S_1)$; and the result type (Sapp) is the *gradual codomain* $\widetilde{cod}(S_1)$ of the operator type. This structure keeps the typing rules syntax-directed, while subsuming the type system for the corresponding statically typed language STLC.

$\boxed{\Gamma \vdash t : S}$ **Gradual Typing**

$$\text{G-Int}\ \frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \text{G-Id}\ \frac{\Gamma(x) = S}{\Gamma \vdash x : S} \qquad \text{G-Lam}\ \frac{[x \mapsto S_1]\Gamma \vdash t : S_2}{\Gamma \vdash \lambda\,(x : S_1)\,.\,t : S_1 \to S_2}$$

$$\text{G-Ann}\ \frac{\Gamma \vdash t : S_1 \qquad S_1 \sim S_2}{\Gamma \vdash t :: S_2 : S_2} \qquad \text{G-App}\ \frac{\Gamma \vdash t_1 : S_1 \qquad \Gamma \vdash t_2 : S_2 \qquad S_2 \sim \widetilde{dom}(S_1)}{\Gamma \vdash t_1\ t_2 : \widetilde{cod}(S_1)}$$

$\boxed{S \sim S}$ **Consistent Type Equality**

$$\frac{}{\mathsf{Int} \sim \mathsf{Int}} \qquad \frac{}{? \sim S} \qquad \frac{}{S \sim ?} \qquad \frac{S_3 \sim S_1 \qquad S_2 \sim S_4}{S_1 \to S_2 \sim S_3 \to S_4}$$

Figure 3.2: Gradual Surface Language for AGTLC

3.1.6 *Evidence objects for the runtime language*

The AGTLC language does not yet have a runtime semantics. In a traditional gradual language design, the next step of design would be to introduce run-time checks in the form of *type casts*. The language designer must devise a semantics for their behaviour (in particular, for their propagation and elimination). AGT does not use casts, but instead it uses *evidence objects*, whose semantics is specified by the AGT recipe.

An evidence object is a run-time representation of a consistent judgment, equipped with a set of partial functions that operate over them. We introduce evidence operations as needed while we explain the design of the runtime language for AGTLC, which we call ARL.

ARL mirrors AGTLC but for a few differences. First, runtime terms have no source typing information: since ARL programs are constructed by type-directed translation from AGTLC (Section 3.1.8), we can presume the existence of a corresponding typing derivation, and we need not reconstruct one by analyzing ARL terms. Omitting static types also lets us formally distinguish type information that is used for static checking from information that is used for run-time enforcement.

Second, many ARL terms, in particular the subterms of elimination forms and ascriptions, are decorated with *evidence* objects ε, which summarize run-time consistent equality judgments. Recall that consistent equality denotes the *plausibility* that a type equality relationship holds between two gradual types. Evidence objects reify that plausibility, and evolve as part of run-time type enforcement.

To collect and enforce all the optimistic assumptions that the gradual source-level type checker made, the AGT recipe guides the design of the syntax and structure of the runtime for the gradual language with a *run-time representation* of the consistent equality proofs. We call this representation *evidence objects*.

The design of the run-time language follows the structure of the static language by weakening equality checks to consistent equality checks, and by relying on evidence objects to guarantee that the static assumptions are consistently enforced throughout

evaluation. Evidence objects should be available whenever the static type system needed to refer to type equality.

Evidence objects ε are intended to *justify* the consistent equality predicate: The run-time type system relies on judgments of the form $\varepsilon \vdash S_0 \sim S_1$, which say that ε contains sufficient information to justify the claim that $S_0 \sim S_1$. The convention in AGT is for the evidence object to be a tuple of gradual types with the same arity as the original static predicate that is being justified, although other designs are also possible. In the case of equality, the evidence objects are pairs of gradual types because the equality predicate takes two arguments.

Not every pair of gradual types can be used as an evidence object: an evidence object needs to be well-formed.

Definition 9 (Well-formed evidence object)**.** *We say an evidence object for consistent equality is* well-formed *if both:*

1. *The gradual types in the evidence object are related by consistent equality ($S_1 \sim S_2$,* i.e. *$T_1 = T_2$ for some $T_1 \sqsubseteq S_1$ and $T_2 \sqsubseteq S_2$).*

2. *The gradual types in the evidence object are the least upper bound over precision among all the static types related by equality. Formally, suppose $S_1', S_2' \in$ GType and whenever $T_1 = T_2$, $T_1 \sqsubseteq S_1$, and $T_2 \sqsubseteq S_2$, we also have $T_1 \sqsubseteq S_1'$ and $T_2 \sqsubseteq S_2'$. Then $S_1 \sqsubseteq S_1'$ and $S_2 \sqsubseteq S_2'$.*

In short, an evidence object is the least (with respect to precision ($\sqsubseteq$)) pair of gradual types that subsumes some set of static equality predicate instances $T_1 = T_2$. In this sense they are minimal, sound representatives of consistent type equality. For instance, Int is consistently equal to ? ($\mathsf{Int} \sim ?$), but $\langle \mathsf{Int}, ? \rangle$ is *not* well-formed evidence, because $\mathsf{Int} = \mathsf{Int}$ is the only pair of static types that justifies equality between these gradual types. The well-formed evidence object $\langle \mathsf{Int}, \mathsf{Int} \rangle$ captures the same information. From the definition of well-formed evidence, it follows immediately that if $\varepsilon \vdash S_1 \sim S_2$ for any ε, then indeed $S_1 \sim S_2$ holds in general.

This well-formed evidence object specification is equivalent to the following inductive definition::

$\boxed{\vdash \varepsilon \ \mathbf{wf}}$

Well-formed Evidence

$$\frac{S \in \{\, \mathsf{Int}, ? \,\}}{\vdash \langle S, S \rangle \ \mathbf{wf}} \qquad \frac{\vdash \langle S_{21}, S_{11} \rangle \ \mathbf{wf} \quad \vdash \langle S_{12}, S_{22} \rangle \ \mathbf{wf}}{\vdash \langle S_{11} \to S_{12}, S_{21} \to S_{22} \rangle \ \mathbf{wf}}$$

A well-formed evidence object can be used to *justify* a consistent equality judgment. We separate the evidence object from the justified judgment so that an evidence object may *gain* precision as evaluation progresses.

We base our definition of consistent type equality justification on well-formed evidence objects, and we use this evidence judgment in the type system for the runtime language in Section 3.1.7.

Evidence that justifies a consistent type equality judgment

$$\frac{\vdash \langle S_1', S_2' \rangle \ \mathbf{wf} \qquad S_1' \sqsubseteq S_1 \qquad S_2' \sqsubseteq S_2}{\langle S_1', S_2' \rangle \vdash S_1 \sim S_2}$$

The *evidence judgment* $\varepsilon \vdash S_1 \sim S_2$ says that ε confirms that $S_1 \sim S_2$ is plausible. Besides requiring the evidence object to be well-formed, the judgment connects each gradual type with one of the types in a consistent equality judgment via precision. As we saw when explaining well-formed evidence objects, not all types that are consistently equal make for well-formed evidence objects, so we must account for some difference. But this design also allows us to increase the precision of an evidence object. For example, we may want to do this to filter out inconsistent information between evidence objects that together justify a transitive check of gradual type equality (as seen in Section 3.1.7).

Why would one want to lessen the imprecision of evidence objects as evaluation progresses? A gradual equality judgment $S_1 \sim S_2$ holds when type equality holds some set of static types in their respective meanings. The intuition behind lessening imprecision on the evidence judgment is to refine and reduce this set of static type equality judgments as we observe new information. Eventually, we could find ourselves with an empty set, and that should lead to a run-time error: An empty set represents having no justification for a consistent equality judgment, so evaluation cannot safely continue.

3.1.7 *Obtaining the gradual runtime language*

What information do we need to keep at run time to safely run programs? The runtime terms represent the typing derivation of some well-typed gradual program.[9] The syntax for the runtime term typing judgment is thus $\mathbf{e} \blacktriangleright \Gamma \vdash t : S$, which is read as " the run-time term $\mathbf{e}$ represents the well-typed gradual program t of type S under context Γ".

The typing judgment for runtime terms is unusual in that it classifies a runtime term $\mathbf{e}$ with respect to a *source typing judgment* $\Gamma \vdash t : S$. The judgment $\mathbf{e} \blacktriangleright \Gamma \vdash t : S$ says that the runtime term $\mathbf{e}$ *represents* a source gradual typing derivation $\Gamma \vdash t : S$, where t is a source-language AGTLC term. The motivation for this structure is that in a gradually typed language, type enforcement is not necessarily completed at type-checking time. Some type enforcement may be deferred to run time, and this enforcement is construed as an attempt to complete (or refute) the type safety argument at run time. Thus one can think of an ARL term's execution as playing out type progress and preservation, which may ultimately fail and justifiably signal a run-time type error. In short, ARL terms represent the computationally relevant residual of a AGTLC typing derivation.

We use the type safety proof from the static type system as a guideline to develop the syntax, typing judgment, and reduction relation for the runtime language. In particular, we care about the *type preservation* proof associated with the static type system. The way we have laid out that static type system makes all dependencies explicit, and we revisit the cases of the proof of type preservation that are relevant as we introduce the parts of the runtime language.

As a reminder, our definition of type preservation for the static language goes in two parts:

Theorem 5 (Type preservation for notions of reduction)**.** *For any well-typed program t that can take a reduction step $(t \rightsquigarrow t')$, then t' is well-typed under the same context as t and at the same type.*

Theorem 6 (Type preservation for contextual reduction)**.** *For any well-typed program t that can take a step $(t \longrightarrow t')$, then t' is well-typed under the same context as t and at the same type.*

To introduce the details of the runtime language, we will follow the structure of the proofs of each of these theorems. We do this by first dealing with all the cases of notions of reduction (in this language, just type ascriptions and function application), and then by dealing with the structural reduction rule.

3.1.7.1 *Dealing with type ascriptions (obtaining transitive proofs)*

A type ascription in the source language depends on a consistent type equality judgment, so the run-time representation should carry an evidence object for that judgment, as shown in the following runtime typing derivation rule:

$$(S::)\frac{\mathbf{e} \blacktriangleright \Gamma \vdash t : S \qquad \varepsilon \vdash S \sim S'}{\varepsilon\mathbf{e} \blacktriangleright \Gamma \vdash (t :: S') : S'}$$

The greyed type S' is not explicitly present in the runtime program, but it must exist to type the rest of the surrounding program. It is justified by the evidence object itself, which justifies the consistent equality check connecting the type of t (that is, S) and the type of the ascription S'.

To motivate the evaluation of ascriptions in the runtime language, let's revisit the proof of type preservation for the reduction of ascriptions in the static language. In the static language, type ascriptions only reduce using the rule $(u :: T_1 :: T_2) \rightsquigarrow (u :: T_2)$. Therefore, the typing derivation of the premise term for type preservation $((u :: T_1) :: T_2)$ can only take the following form:

$$\text{Ann}\frac{\text{Ann}\dfrac{\begin{array}{c}\mathcal{D}_u\\ \vdots\\ \Gamma \vdash u : T_0\end{array} \qquad \begin{array}{c}\mathcal{D}_{0=}\\ \vdots\\ T_0 = T_1\end{array}}{\Gamma \vdash u :: T_1 : T_1} \qquad \begin{array}{c}\mathcal{D}_{1=}\\ \vdots\\ T_1 = T_2\end{array}}{\Gamma \vdash (u :: T_1) :: T_2 : T_2}$$

That is, there is some typing derivation $\mathcal{D}_u$ saying that the value u has type T_0, and two derivations $\mathcal{D}_{0=}$ and $\mathcal{D}_{1=}$ saying that two types are equal, all combined via applications of the inference rule [Ann]. To prove that the value after reduction also has type T_2, we build a new typing derivation at the same type. Building this derivation takes steps that may seem obvious, but that we want to make explicit as they also guide evaluation in the runtime language:

1. We can reuse the sub-derivation $\mathcal{D}_u$ to conclude that u has type T_0.

2. We can combine derivations $\mathcal{D}_{0=}$ and $\mathcal{D}_{1=}$ by *transitivity of equality* to conclude that T_0 is equal to T_2.

3. We can use these two derivations to build a new typing derivation for $(u :: T_2)$ using inference rule [Ann].

The key step here is that, although the type T_1 has completely disappeared from the evaluated program, it remains implicitly present in the proof: we had to combine the equality judgments that involved type T_1 to build the appropriate proof of transitivity of equality that is needed in the new typing derivation.

We perform the same process to reduce type ascriptions in the gradual language. Intuitively, we do so for every concrete static type in the meaning of the gradual types at hand. We combine evidence objects to represent the result of combining equality proofs via transitivity. This combination requires a partial function from two evidence objects into a new one, called *evidence composition*.

Evidence composition $\varepsilon_1 \fatsemi \varepsilon_2$ is used to monitor *transitivity* of consistent equality. For this reason, this composition operation is also called *consistent transitivity* (Garcia, Clark, and Tanter, 2016). The defining correctness criteria for consistent transitivity are as follows.

Definition 10 (Candidates for transitivity)**.** *Suppose two evidence judgments that share a supported gradual type "in the middle", formally $\varepsilon_1 = \langle S'_1, S'_{21} \rangle \vdash S_1 \sim S_2$ and $\varepsilon_2 = \langle S'_{22}, S'_3 \rangle \vdash S_2 \sim S_3$ (note that S_2 appears on both judgments in different locations, which we call "the middle" of the transitive path from S_1 to S_3).*

1. *We then call (ε_1,ε_2)* plausibly transitive *if we can form a transitive chain of three static types related by equality, each less imprecise than the type in the appropriate portion of the evidence objects, while reusing the same type in the middle for the portions of both evidence objects that were also "in the middle". Formally, $T_1 \sqsubseteq S'_1$ and $S'_{21} \sqsupseteq T_2 \sqsubseteq S'_{22}$, and $T_3 \sqsubseteq S'_3$ hold for some triple $T_1 = T_2 = T_3$;*

2. *We then call $\varepsilon' = \langle S''_1, S''_3 \rangle$ a* candidate for transitivity of (ε_1,ε_2) *if (ε_1,ε_2) is plausibly transitive, and if for every triple of static types that could be used to justify (ε_1,ε_2) as plausibly transitive, the types on the extremes are also bound via precision to the respective type in ε'. Formally, every triple $T_1 = T_2 = T_3$ such that $T_1 \sqsubseteq S'_1$ and $S'_{21} \sqsupseteq T_2 \sqsubseteq S'_{22}$, and $T_3 \sqsubseteq S'_3$ implies $T_1 \sqsubseteq S''_1$ and $T_3 \sqsubseteq S''_3$;*

Proposition 7 (Consistent transitivity or evidence composition)**.** *Suppose two evidence judgments that share a supported gradual type "in the middle", formally $\varepsilon_1 = \langle S'_1, S'_{21} \rangle \vdash S_1 \sim S_2$ and $\varepsilon_2 = \langle S'_{22}, S'_3 \rangle \vdash S_2 \sim S_3$. Then ($\varepsilon_1$,$\varepsilon_2$) has a candidate for transitivity if and only if it has a least candidate over the precision ($\sqsubseteq$) order, denoted $\varepsilon_1 \fatsemi \varepsilon_2$. Furthermore, this least candidate also supports consistent equality among the types on the extremes, formally $(\varepsilon_1 \fatsemi \varepsilon_2) \vdash S_1 \sim S_3$.*

As Siek and Taha (2006) first observed, consistent equality is not transitive in general, and that property is fundamental to gradual type checking: The gradual language accepts many programs which, after being transitively reduced, reveal an inconsistency that should be rejected and which leads evaluation to fail with a run-time error. However, transitivity of equality is fundamental to proving type safety. Thus, like in all AGT-based languages, run-time type errors in AGTLC distill down to a failure of transitivity:

consistent transitivity is a *partial* function, and when it is undefined, we interpret the result as losing any justification to proceed with evaluation and we produce a run-time error instead.

Consistent transitivity lets us transform the notion of reduction for type annotations from the static language into the gradual language. We move from the rule $(u :: T_1) :: T_2 \rightsquigarrow (u :: T_2)$ to two rules, depending on whether evidence composition succeeds or fails:

$$\begin{array}{lcll} \varepsilon_2(\varepsilon_1 u) & \rightsquigarrow & \varepsilon_3 u & \text{(where } \varepsilon_3 = \varepsilon_1 \fatsemi \varepsilon_2\text{)} \\ \varepsilon_2(\varepsilon_1 u) & \rightsquigarrow & \textbf{error} & \text{(if } \varepsilon_1 \fatsemi \varepsilon_2 \text{ is undefined)} \end{array}$$

3.1.7.2 *Dealing with function application (obtaining inversion principles)*

Reducing function applications is more complex than reducing type ascriptions, because transitivity of type equality is not the only extra operation involved in the original proof of static type preservation. The proof also relies on two other partial functions over evidence objects, *idom* and *icod*.

To motivate the evaluation of function applications in the runtime language, let's first revisit the proof of type preservation for function application in the static type language. In the static language, function applications only reduce using the rule $(\lambda\,(x : T)\,.\,t)\;\; v \rightsquigarrow [v/x]\,t$. Therefore, the typing derivation of the premise term for type preservation $((\lambda\,(x : T)\,.\,t)\;\; v)$ can only take the following form:

$$\text{App}\,\dfrac{\begin{array}{c}\mathcal{D}_\lambda \\ \vdots \\ \Gamma \vdash \lambda\,(x : dom(T_0))\,.\,t : T_0\end{array} \quad \begin{array}{c}\mathcal{D}_v \\ \vdots \\ \Gamma \vdash v : T_2\end{array} \quad \begin{array}{c}\mathcal{D}_{0=} \\ \vdots \\ T_2 = T_1\end{array} \quad \begin{array}{c}\mathcal{D}_{1=} \\ \vdots \\ T_0 = T_1 \to T_3\end{array}}{\Gamma \vdash (\lambda\,(x : dom(T_0))\,.\,t)\;\; v : T_3}$$

The proof that this reduction preserves types goes as follows:

- By inversion on the second equality judgment, we know that $T_1 = dom(T_0)$ and that $cod(T_0) = T_3$, obtaining some proofs for each ($\mathcal{D}_{dom=}$ and $\mathcal{D}_{cod=}$, respectively).
- By a separate lemma, substitution preserves the type of t (that is, $cod(T_0)$) when given a value at the right type (that is, $dom(T_0)$). To ensure that the argument has the right type, we combine derivation $\mathcal{D}_{0=}$ with the derivation $\mathcal{D}_{dom=}$ we just obtained to conclude that T_2 is equal to $dom(T_0)$ by *transitivity of equality*.
- To conclude we have preserved the type of the full derivation, S_3, we rely on the fact that $cod(S_0)$ is equal to S_3 coming from derivation $\mathcal{D}_{cod=}$.

For any appeal to transitivity of equality, we can reuse the evidence composition operation we introduced for type ascriptions. We still need some way to provide a run-time representation of *inversion principles* for consistent equality over evidence objects. For this purpose we introduce the *idom* and *icod* operations.

Lets first consider inversion for the codomain operations, which we call *icod*. Given the inductive definition of consistent equality in Figure 3.2, we can prove the following inversion lemma:

Lemma 1. *Consider two gradual types S_1 and S_2 related by consistent equality. If the codomain of the type on the right is defined, then the codomain of the type on the left is also defined, and $cod(S_1) \sim cod(S_2)$*

The *icod* operator reifies this inversion principle at run time: If $\varepsilon \vdash S_1 \sim S_2$ then $icod(\varepsilon) \vdash \widetilde{cod}(S_1) \sim \widetilde{cod}(S_2)$. So just as inversion principles are used to prove static type safety, evidence inversion operators are used to extract inversion information at run time. A similar process can be followed for the *idom* operation with respect to $\widetilde{dom}$. These inversion operators are *total* functions over appropriate subdomains of evidence, which properly reflect the kind of run-time evidence that they manipulate during reduction. As such, inversion never fails.

We can define these inversion operations as follows:

Inversion Functions

$$\textsc{FunEv} = \{\, \varepsilon \in \textsc{Ev} \mid \varepsilon \vdash S_1 \sim S_{21} \to S_{22} \,\}$$

$$\begin{array}{ll} idom : \textsc{FunEv} \to \textsc{Ev} & icod : \textsc{FunEv} \to \textsc{Ev} \\ idom(S_1, S_2) = \langle \widetilde{dom}(S_2), \widetilde{dom}(S_1) \rangle & icod(S_1, S_2) = \langle \widetilde{cod}(S_1), \widetilde{cod}(S_2) \rangle \end{array}$$

We use these inversion operations and consistent transitivity to transform the original evaluation rule into the following two, depending on whether evidence composition succeeds or fails:

$$\begin{array}{lcll} \varepsilon_1\ (\lambda\,(\mathbf{x})\,.\,\mathbf{e})\ \ \varepsilon_2\mathbf{v} & \rightsquigarrow & icod(\varepsilon_1)\,[\varepsilon_{\mathbf{v}}\mathbf{v}/x]\,\mathbf{e} & (\text{where } \varepsilon_{\mathbf{v}} = \varepsilon_2 \fatsemi idom(\varepsilon_1)) \\ \varepsilon_1\ (\lambda\,(\mathbf{x})\,.\,\mathbf{e})\ \ \varepsilon_2\mathbf{v} & \rightsquigarrow & \textbf{error} & \begin{array}{l}(\text{if } \varepsilon_2 \fatsemi idom(\varepsilon_1) \\ \quad \text{is undefined})\end{array} \end{array}$$

These operations are justified by the proof of static type preservation we previously discussed.

To allow for these reduction rules, we introduce extra evidence objects to the language, leading to the following definition of syntax for runtime values:

Syntax

$$\begin{array}{rcll} \varepsilon \in \textsc{Ev} & = & \{\, \langle S_1, S_2 \rangle \mid \vdash \langle S_1, S_2 \rangle\ \ \mathbf{wf} \,\} & \text{(evidence objects)} \\ \mathbf{e} \in \textsc{RTerm} & ::= & \mathbf{n} \mid \mathbf{x} \mid \lambda\mathbf{x}.\mathbf{e} \mid \varepsilon\mathbf{e}\ \varepsilon\mathbf{e} \mid \varepsilon\mathbf{e} & \text{(runtime terms)} \\ \mathbf{u} \in \textsc{RawValue} & ::= & \mathbf{n} \mid \lambda\mathbf{x}.\mathbf{e} & \text{(raw values)} \\ \mathbf{v} \in \textsc{Value} & ::= & \mathbf{u} \mid \varepsilon\mathbf{u} & \text{(values)} \end{array}$$

We introduce the complete runtime type system for ARL in Figure 3.3.

3.1.7.3 *Dealing with contextual reduction*

The STLC language allows for reduction to happen in subexpressions via the contextual reduction relation. This relation depends on a definition of evaluation contexts that we must be able to translate to the runtime language.

Evaluation contexts for the runtime language need to be slightly different. Because we are following the proof of type preservation, we need to perform reduction on the typing derivations of sub-expressions, which do not include the evidence object that connects it to the broader runtime typing derivation. That evidence object needs to wait on the stack, leading to the following definition of evaluation contexts:

Syntax

$$\begin{array}{rcll} E \in \textsc{ECtxt} & ::= & \square \mid E[F[\varepsilon\square]] & \text{(evaluation contexts)} \\ F \in \textsc{EvFrame} & ::= & \square \mid \square\ \varepsilon\mathbf{e} \mid \varepsilon\mathbf{u}\ \square & \text{(evidence frame)} \end{array}$$

This introduces a small difficulty: Although we have dealt with nested type ascriptions in the notions of reduction, we may be left with extra evidence objects connecting subexpressions that are not exposed to the internal notions of reduction. The solution to this problem is to copy the notions of reduction for ascription to also be allowed directly at the level of contextual reduction. Since this new placing subsumes the original, we can also remove the original notion of reduction for ascriptions, obtaining the following definition for the runtime semantics:

$\boxed{\mathbf{e} \rightsquigarrow \mathbf{e},\ \mathbf{e} \rightsquigarrow \mathbf{error}}$

Notions of Reduction

$$\varepsilon_1(\lambda\mathbf{x}.\mathbf{e})\ \varepsilon_2\mathbf{u} \rightsquigarrow \begin{cases} \big(icod(\varepsilon_1)\big)\big([(\varepsilon_{\mathbf{u}}\mathbf{u})/\mathbf{x}]\mathbf{e}\big) & \text{(where } \varepsilon_{\mathbf{u}} = \varepsilon_2 \fatsemi idom(\varepsilon_1)\text{)} \\ \mathbf{error} \quad \text{if } \varepsilon_{\mathbf{u}} \text{ not defined} & \end{cases}$$

$\boxed{\mathbf{e} \longrightarrow \mathbf{e},\ \mathbf{e} \longrightarrow \mathbf{error}}$

Contextual Reduction

$$\frac{\mathbf{e} \rightsquigarrow \mathbf{e}'}{E[\mathbf{e}] \longrightarrow E[\mathbf{e}']} \qquad \frac{\mathbf{e} \rightsquigarrow \mathbf{error}}{E[\mathbf{e}] \longrightarrow \mathbf{error}}$$

$$\frac{\varepsilon_2 \fatsemi \varepsilon_1 \text{ not defined}}{E[F[\varepsilon_1\varepsilon_2\mathbf{u}]] \longrightarrow \mathbf{error}} \qquad \frac{}{E[F[\varepsilon_1\varepsilon_2\mathbf{u}]] \longrightarrow E[F[(\varepsilon_2 \fatsemi \varepsilon_1)\mathbf{u}]]}$$

3.1.7.4 *Type safety of ARL*

ARL satisfies standard type safety (Wright and Felleisen, 1994) with respect to the type system in Figure 3.3: well-typed terms are either reducible using the semantics or they are in a canonical form. Programs do not get stuck, but they may signal run-time type errors.[10] Progress and preservation are sufficient for type safety: Together they imply that terms do not get stuck and that every intermediate step has the same type.

$\boxed{\mathbf{e} \blacktriangleright \Gamma \vdash t : S}$ **Runtime Typing**

$$(Sn)\frac{}{\mathbf{n} \blacktriangleright \Gamma \vdash n : \mathsf{Int}} \qquad (S::)\frac{\mathbf{e} \blacktriangleright \Gamma \vdash t : S \quad \varepsilon \vdash S \sim S'}{\varepsilon\mathbf{e} \blacktriangleright \Gamma \vdash (t :: S') : S'}$$

$$(S\mathrm{app})\frac{\overline{\mathbf{e}_i \blacktriangleright \Gamma \vdash t_i : S_i} \quad \varepsilon_1 \vdash S_1 \sim S_1' \to S_2' \quad \varepsilon_2 \vdash S_2 \sim S_1'}{(\varepsilon_1\mathbf{e}_1\ \varepsilon_2\mathbf{e}_2) \blacktriangleright \Gamma \vdash t_1\ t_2 : S_2'}$$

$$(Sx)\frac{x : S' \in \Gamma}{\mathbf{x} \blacktriangleright \Gamma \vdash x : S'} \qquad (S\lambda)\frac{\mathbf{e} \blacktriangleright \Gamma, x : S_1 \vdash t : S_2}{(\lambda x.\mathbf{e}) \blacktriangleright \Gamma \vdash (\lambda x : S_1.t) : S_1 \to S_2}$$

Figure 3.3: Runtime type system for ARL

Proposition 8 (Progress)**.** *If a runtime term* $\mathbf{e}$ *represents a derivation that some term* t *is well-typed under a closed context* $\mathbf{e} \blacktriangleright \emptyset \vdash t : S$*, then one of the following holds:*

1. $\mathbf{e}$ *is a value* $\mathbf{v}$*;*
2. $\mathbf{e}$ *can take a reduction step to a new runtime term,* $\mathbf{e} \to \mathbf{e}'$*;*
3. $\mathbf{e}$ *can take a reduction step to an error,* $\mathbf{e} \to \mathbf{error}$*.*

Proposition 9 (Preservation)**.** *If a runtime term represents some well-typed derivation under a closed context and the term can take a step to a new term (i.e., not an error), the new term also represents some (other) well-typed derivation under a closed context and both well-typed terms have the same type. Formally, if* $\mathbf{e} \blacktriangleright \emptyset \vdash t : S$ *and* $\mathbf{e} \to \mathbf{e}'$ *then* $\mathbf{e}' \blacktriangleright \emptyset \vdash t' : S$ *for some source* t'*.*

The statement of type preservation is unusual, because the runtime typing judgment expresses a crisp relationship between source terms and runtime terms. The source terms t in the runtime typing judgment evolve in lock-step with the runtime terms $\mathbf{e}$. We say more about this in Section 3.1.8.

3.1.8 *Obtaining a translation from the surface language into the runtime (elaboration)*

AGTLC source programs are elaborated to ARL programs by type-directed translation (Figure 3.4). The elaboration process is quite uniform, exhibiting the tight connection between AGTLC and ARL. In essence, ARL terms represent AGTLC derivation trees, adding essential run-time type information (*i.e.*, evidence) and erasing superfluous source type information.

$\boxed{\Gamma \vdash t \rightsquigarrow \mathbf{e} : S}$ **Elaboration**

$$(\rightsquigarrow\text{x})\frac{x : S \in \Gamma}{\Gamma \vdash x \rightsquigarrow \mathbf{x} : S} \qquad (\rightsquigarrow\text{n})\frac{}{\Gamma \vdash n \rightsquigarrow \mathbf{n} : \mathsf{Int}}$$

$$(\rightsquigarrow\text{app})\frac{\begin{array}{ll}\Gamma \vdash t_1 \rightsquigarrow \mathbf{e}_1 : S_1 & \varepsilon_1 = \mathcal{I}[\![S_1 \sim \widetilde{dom}(S_1) \to \widetilde{cod}(S_1)]\!] \\ \Gamma \vdash t_2 \rightsquigarrow \mathbf{e}_2 : S_2 & \varepsilon_2 = \mathcal{I}[\![S_2 \sim \widetilde{dom}(S_1)]\!]\end{array}}{\Gamma \vdash t_1\ t_2 \rightsquigarrow \varepsilon_1\mathbf{e}_1\ \varepsilon_2\mathbf{e}_2 : \widetilde{cod}(S_1)}$$

$$(\rightsquigarrow\lambda)\frac{\Gamma, x : S_1 \vdash t \rightsquigarrow \mathbf{e} : S_2}{\Gamma \vdash (\lambda x : S_1.t) \rightsquigarrow (\lambda \mathbf{x}.\mathbf{e}) : S_1 \to S_2} \qquad (\rightsquigarrow::)\frac{\Gamma \vdash t \rightsquigarrow \mathbf{e} : S \quad \varepsilon = \mathcal{I}[\![S \sim S_1]\!]}{\Gamma \vdash (t :: S_1) \rightsquigarrow \varepsilon\mathbf{e} : S_1}$$

Figure 3.4: AGTLC: Elaboration

The design of the translation follows the structure of the well moded static typing derivation. For example, consider the transformation that begins from a typing derivation with the following structure:

$$\text{G-App}\frac{\begin{array}{ccc}\mathcal{D}_1 & \mathcal{D}_2 & \\ \vdots & \vdots & S_2 \sim dom(S_1) \\ \Gamma \vdash t_1 : S_1 & \Gamma \vdash t_2 : S_2 & \end{array}}{\Gamma \vdash t_1\ t_2 : cod(S_1)}$$

To transform this derivation, we recursively transform source-level derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ into new runtime-level derivations $\mathcal{D}_3$ and $\mathcal{D}_4$, and then build the required type equality constraints for the runtime type system with the information available, ending up with the following typing derivation for a runtime term, with $\mathbf{e}_1$ and $\mathbf{e}_2$ being the runtime terms resulting from recursively translating t_1 and t_2:

$$(\text{Sapp})\frac{\begin{array}{ccc}\mathcal{D}_3 & \mathcal{D}_4 & \\ \vdots & \vdots & \varepsilon_1 \vdash S_2 \sim dom(S_1) \\ \mathbf{e}_1 \blacktriangleright \Gamma \vdash t_1 : S_1 & \mathbf{e}_2 \blacktriangleright \Gamma \vdash t_2 : S_2 & \varepsilon_2 \vdash S_1 \sim dom(S_1) \to cod(S_1)\end{array}}{\left(\varepsilon_1\ \mathbf{e}_1\ \varepsilon_2\ \mathbf{e}_2\right) \blacktriangleright \Gamma \vdash t_1\ t_2 : cod(S_1)}$$

This derivation needs two evidence objects, ε_1 and ε_2, which seem to come out of nowhere. Injecting a program from the gradual surface language into a runtime program configuration requires an operation that can generate evidence objects ε_1 and ε_2 from their consistent equality judgments. Once we can generate evidence objects for these predicates, we can mirror this exact translation into the elaboration rule ($\rightsquigarrow$app) in Figure 3.4.

Each elaboration rule corresponds directly to a source AGTLC typing rule and its translation into the run-time configuration type system. In particular, each premise typing judgment becomes a corresponding elaboration judgment, and each consistent equality judgment introduces an evidence object, using the *initial evidence* operator.[11]

The initial evidence operator $\mathcal{I}[\![S_1 \sim S_2]\!]$ computes the *most imprecise* evidence object $\varepsilon \equiv \langle S_1', S_2' \rangle$, that justifies the consistent equality predicate $S_1 \sim S_2$ (which we write as

$\langle S_1', S_2' \rangle \vdash S_1 \sim S_2$). We want the most imprecise evidence object as it manifests the least possible commitment given the evidence abstraction and the desire to justify $S_1 \sim S_2$. Evidence objects may gain precision at run time, but we want to start out being as accommodating as possible within the gradual typing constraints imposed in the source program. From the definition of well-formed evidence in Section 3.1.6, we can deduce that the initial evidence object $\langle S_1', S_2' \rangle$ is also the *least imprecise* pair of gradual types that is an upper bound over precision for every static type equality instance less imprecise than the consistent equality predicate the operator starts from. That is, for any pair of static types such that $T_1 = T_2$, then $\langle T_1, T_2 \rangle \sqsubseteq \langle S_1, S_2 \rangle$ if and only if also $\langle T_1, T_2 \rangle \sqsubseteq \langle S_1', S_2' \rangle$.

Initial evidence is undefined when the types are not related by consistent equality ($S_1 \not\sim S_2$). This is a helpful feature, as type checking depends only on consistent equality and not on its negation. Therefore whenever type checking succeeds, initial evidence is also defined.

The tight connection between AGTLC and ARL is confirmed by preservation of well-typedness.

Proposition 10 (Well-formed Translation)**.** *The runtime term resulting from elaboration represents the typing derivation that guided the elaboration process. Formally, if* $\Gamma \vdash t \rightsquigarrow \mathbf{e} : S$ *then* $\mathbf{e} \blacktriangleright \Gamma \vdash t : S$.

Because the translation is well-formed, the source term t elaborates to a runtime term $\mathbf{e}$ that represents t's static typing derivation.

We are now equipped to better explain the statement of preservation in Proposition 9. We relate runtime terms to source terms using the runtime typing judgment $\mathbf{e} \blacktriangleright \Gamma \vdash t : S$ and source terms to runtime terms using the elaboration judgment $\Gamma \vdash t \rightsquigarrow \mathbf{e} : S$. Preservation clarifies how runtime terms *"learn"* new type constraints that are not evident in source programs.

Proposition 11 (Replicant)**.** *If* $\mathbf{e} \blacktriangleright \Gamma \vdash t : S$ *and* $\Gamma \vdash t \rightsquigarrow \mathbf{e}' : S$ *then* $\mathbf{e} \sqsubseteq \mathbf{e}'$.

In this proposition, $\mathbf{e} \sqsubseteq \mathbf{e}'$ refers to the pointwise extension of evidence precision from types to terms. This proposition says that a runtime term $\mathbf{e}$ embodies at least as many run-time type constraints as any source term that it represents. In the above term, we call $\mathbf{e}'$ a *replicant* of $\mathbf{e}$, because it amounts to cloning the structure of $\mathbf{e}$ but omitting its *"memory"* of any additional type constraints acquired while evaluating the program. The $\mathbf{e}'$ term starts with a clean slate, yet to be jaded by the tribulations of run-time type enforcement.

The key idea is that a source program contains enough local type information to justify its *plausible* typeability, but the runtime term $\mathbf{e}$ may account for type invariants revealed at run time. In Section 3.2.5, we observe that these discovered type constraints must be represented precisely in order to enforce type-based invariants at run time in mixed-type programs.

3.1.9 *Comparing AGTLC and GTLC*

Now that we have successfully obtained a gradual language using the AGT recipe, we can compare it with previous *ad hoc* approaches. The GTLC of Siek, Vitousek, et al. (2015)

and AGTLC share a static semantics: The same programs get rejected by the type checker. They do not have the same runtime semantics, although the differences are subtle.

As mentioned in Chapter 2, one way to compare different semantics for gradual typing is with respect to how many programs produce run-time errors via the notion of "Err-approximation" introduced by Greenman and Felleisen (2018). AGTLC is strictly more sensitive to type inconsistencies than the GTLC or "Natural" semantics for gradual typing. In particular, although GTLC always errors out when first-order casts are inconsistent, this error-detection behaviour does not transfer up to always-inconsistent functions in GTLC. On the other hand, AGTLC can detect always-inconsistent function type annotations.

Theorem 12 (AGTLC Err-approximates Natural)**.** *For any program that* $\mathbf{e} \blacktriangleright \Gamma \vdash t : S$, $[\![\mathbf{e} \blacktriangleright \Gamma \vdash t : S]\!]_{Natural} \longrightarrow^*_{Natural} \mathbf{error}$ *implies that* $\mathbf{e} \longrightarrow^* \mathbf{error}$.

Proof. The proof is available in Appendix B. □

Theorem 13 (AGTLC is strictly more error sensitive than Natural)**.** *There exists a program such that* $\mathbf{e} \blacktriangleright \Gamma \vdash t : S$, $\mathbf{e} \longrightarrow \mathbf{error}$ *and* $[\![\Gamma \vdash t : S]\!]_{Natural} \longrightarrow v$.

Proof. Consider the following program:

$$
\begin{array}{l}
\mathsf{let}\ f : ? := \lambda\,(x : \mathsf{Int})\,.\,x\ \mathsf{in} \\
\quad \mathsf{let}\ g : ((? \to ?) \to ?) := f\ \mathsf{in} \\
\quad\quad g
\end{array}
$$

The annotations in this program pass through the unknown type but are always inconsistent, thus evidence composition fails before producing a value, aborting execution. On the other hand, GTLC's Natural semantics generates a closure (which, if used, would fail). □

After having familiarized ourselves with the AGT recipe, we can revisit the original AGT language and discuss its issues.

3.2 GTFL$_{\lesssim}$: A GRADUAL LANGUAGE WITH RECORDS AND SUBTYPING

This section presents the semantics of GTFL$_{\lesssim}$, a gradually typed language with records and subtyping that also supports migration between dynamic and static type checking. Garcia, Clark, and Tanter (2016) developed this language using the AGT methodology, and its semantics exhibits the shortcomings that this dissertation chapter addresses.

We present GTFL$_{\lesssim}$ with little reference to the AGT machinery used to construct and justify it. That machinery is discussed when introducing AGTLC in Section 3.1. In this section, we state correctness properties without proof, because AGT exploits calculational abstract interpretation techniques (P. Cousot and R. Cousot, 1977) to intertwine the proof and definition processes, making the design "correct by construction." Ultimately, the improvements to GTFL$_{\lesssim}$ discussed in the following sections are presented in terms of AGT to ensure that they generalize across AGT-based languages.

3.2.1 *The static language we begin from*

The GTFL$_{\lesssim}$ gradual language is the result of applying the AGT recipe to the statically typed STFL$_{<:}$ language, introduced by Garcia, Clark, and Tanter (2016). STFL$_{<:}$ is the simply typed lambda calculus we presented in Section 3.1, extended with records, a subtyping relation (denoted $<:$), and booleans. Records are a data structure mapping labels to values, and a record type maps labels to types. Records also provide a projection operation to extract the value bound to a particular label in a record.

The subtyping relation for records allows the type checker to accept more programs, as many types are not exactly equal but their constraints are compatible. We can compare two record types via subtyping by comparing the types associated to each field. Subtyping can arise by depth when both records share a field for which the type of the first is a subtype of the second. Subtyping can also arise by width when a subtype record has a field that is not present in the supertype. This definition of record subtyping is standard (e.g. Benjamin C. Pierce (2002)).

Another quite useful feature in STFL$_{<:}$ is booleans with if expressions. The type system for if expressions in STFL$_{<:}$ allows for different types to appear in each branch of an if, the only restriction is that both types must share a common supertype. We can obtain the shared supertype with the static join ($\dot{\vee}$) partial function over static types: for example, $\mathsf{Int} \dot{\vee} \mathsf{Bool}$ is undefined because no type in this language is a supertype of both, but the records $[l : \mathsf{Int}]$ and $[l : \mathsf{Bool}]$ do share as a common supertype the empty record $[]$ even though the label they share maps to incompatible types on each.

I present an algorithmic static type system for STFL$_{<:}$ in Figure 3.5, ready for the application of the AGT recipe. Many of the typing rules for STFL$_{<:}$ are the same as for STLC after replacing references to the type equality relation with the subtyping relation. The last addition in STFL$_{<:}$ that differentiates the language from STLC is that we can do something with numbers: there is an addition operation, which shares its impact on the language design with many other primitive operations among atomic types.

3.2.2 *Syntax and Typing*

Figures 3.6 and 3.7 present the GTFL$_{\lesssim}$ syntax and type system. Its terms are typical: numbers, booleans, functions, records, and type ascriptions. All of the novelty lies in its type structure, where common static types—atomics, functions, and records—are augmented with two gradual type constructs that denote imprecise type information. The now-standard *unknown type* $?$ denotes the complete omission of type information (Siek and Taha, 2006). The *gradual row type* $[\overline{l : S}, ?]$, on the other hand, represents a record type with incomplete *field* information. It surely constrains the list $\overline{l}$ of fields with corresponding (gradual) types $\overline{S}$, but the *gradual row designator* $?$ denotes the possibility of additional statically unknown fields.[12] Gradual rows are somewhat analogous to polymorphic rows (Rémy, 1989; Wand, 1991), except that their presence induces dynamic checks. As such, a gradual row type is only partially static with respect to record fields.

Because gradual records and rows are structurally similar, I sometimes use $*$ as a designator marker to signify that some definition applies both to gradual record and gradual row types. I also use $*$ to match the designator of a particular type (as in $*_1$,

$\boxed{\Gamma \vdash t : T}$ **Static Typing**

$$(T\text{x})\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad (T\text{n})\frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad (T\text{b})\frac{}{\Gamma \vdash b : \mathsf{Bool}}$$

$$(T\text{app})\frac{\left(\Gamma \vdash t_i : T_i\right)_{i \in \{1,2\}} \qquad T_2 <: dom(T_1)}{\Gamma \vdash t_1\ t_2 : cod(T_1)}$$

$$(T\text{+})\frac{\left(\Gamma \vdash t_i : T_i\right)_{i \in \{1,2\}} \qquad \left(T_i <: \mathsf{Int}\right)_{i \in \{1,2\}}}{\Gamma \vdash t_1 + t_2 : \mathsf{Int}}$$

$$(T\text{if})\frac{\left(\Gamma \vdash t_i : T_i\right)_{i \in \{1,2,3\}} \qquad T_1 <: \mathsf{Bool}}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T_2 \ddot{\vee} T_3} \qquad (T\text{proj})\frac{\Gamma \vdash t : T}{\Gamma \vdash t.l : proj(T, l)}$$

$$(T\lambda)\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (\lambda x : T_1.t) : T_1 \to T_2} \qquad (T{::})\frac{\Gamma \vdash t : T \qquad T <: T_1}{\Gamma \vdash (t :: T_1) : T_1}$$

$$(T\text{rec})\frac{\left(\Gamma \vdash t_i : T_i\right)_{0 \leq i \leq n}}{\Gamma \vdash [\overline{l_i = t_i}] : [\overline{l_i : T_i}]}$$

Figure 3.5: The static type system of the STFL$_{<:}$ language.

$*_2$, etc.) in an inference rule, which could be either a gradual row or a gradual record type. Examples of this marker can be found in the definition of the $\widetilde{proj}$ partial function in Figure 3.6 and in some inference rules in Figure 3.7.

As discussed in Section 3.1.2, type precision can be defined either in terms of concretization or directly as a relation over gradual types. As we defined precision in terms of concretization for AGTLC, we take the opportunity to follow the alternative approach here and define precision directly for GTFL$_{\lesssim}$ in Figure 3.7.

The *consistent subtyping* relation $\lesssim$ (Siek and Taha, 2007) extends static subtyping $<:$ to optimistically account for imprecision in gradual types. In essence, $S_1 \lesssim S_2$ means that it is *plausible* that S_1 is a subtype of S_2, when the imprecision of gradual types is taken into account. For instance, ? is both a consistent supertype and consistent subtype of each gradual type S, because it could represent any static type whatsoever, including S itself. Following the AGT recipe, we formally relate consistent subtyping to static subtyping

Proposition 14. *A gradual type S_1 is a consistent subtype of a gradual type S_2 if and only if there is a pair of static types that is less imprecise than each of the gradual types, and these two static types are related by static subtyping. Formally, $S_1 \lesssim S_2$ if and only if $T_1 <: T_2$ for some $T_1 \sqsubseteq S_1$ and $T_2 \sqsubseteq S_2$.*

One key benefit to defining gradual operators and relations using static counterparts and gradual type precision is that the resulting language naturally satisfies static criteria for gradual typing set forth by Siek, Vitousek, et al. (2015). First, the STFL$_{<:}$ type system can be recovered from that of GTFL$_{\lesssim}$ by simply restricting source programs to only mention static types T. Doing so restricts the $\widetilde{dom}$, $\widetilde{cod}$, and $\widetilde{proj}$ partial functions to simple arrow type and record type destructors; restricts the *consistent subtyping* relation $\lesssim$ to a typical definition of static subtyping $<:$; and restricts the $\widetilde{\vee}$ partial function to the subtype join partial function $\dot{\vee}$, which yields the least upper bound of two static types (if there is one) according to static subtyping $<:$. Thus, by construction, GTFL$_{\lesssim}$ conservatively extends the static language. Furthermore, the *static gradual guarantee*, follows from the correctness criteria for each operator and relation, each of which monotonically preserves this property, yielding a direct compositional proof.

Though the syntax of gradual rows is simple, its implications for the language semantics are nontrivial. Gradual rows expose a subtle interplay between gradual type precision and static width subtyping that does not arise in most gradual type systems developed to date. Consider $[x : \mathsf{Int}, y : \mathsf{Bool}] \lesssim [x : \mathsf{Int}, ?]$: This judgement is justified by two distinct instances of static subtyping: $[x : \mathsf{Int}, y : \mathsf{Bool}] <: [x : \mathsf{Int}]$ and $[x : \mathsf{Int}, y : \mathsf{Bool}] <: [x : \mathsf{Int}, y : \mathsf{Bool}]$. If this instance of consistent subtyping is viewed as a form of coercion, then it indicates *two* different behaviours: in the first, the y field is *obscured* via static subtyping. In the second, the y field is obscured via precision: not by static subtyping, but by gradual typing. The static *and* runtime semantics must reckon with these two *different* explanations simultaneously, and all outcomes must be consistent with one, the other, or both. This happens in no other language featuring consistent subtyping and can be improved upon with a more precise abstraction. The abstraction of Garcia, Clark, and Tanter (2016) sometimes loses information at run time, which gets obscured due to loss of precision (see next paragraph). By contrast, the BRR abstraction introduced in Section 3.4 preserves all obscured information at run time, but hides some

Syntax

$$x \in \textsc{Var}, \quad b \in \textsc{Bool}, \quad n \in \mathbb{Z}, \quad l \in \textsc{Label}, \quad \Gamma \in \textsc{Env} = \textsc{Var} \xrightarrow{\text{fin}} \textsc{Type}$$

$$\begin{array}{rcll}
T \in \textsc{Type} & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid T \to T \mid [\overline{l : T}] & \text{(static types)} \\
S \in \textsc{GType} & ::= & ? \mid \mathsf{Int} \mid \mathsf{Bool} \mid S \to S \mid [\overline{l : S}] \mid [\overline{l : S}, ?] & \text{(gradual types)} \\
t \in \textsc{Term} & ::= & n \mid b \mid x \mid \lambda x : S.t \mid t\ t \mid t + t \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t & \text{(terms)} \\
 & & \mid [\overline{l = t}] \mid t.l \mid t :: S &
\end{array}$$

$\boxed{\Gamma \vdash t : S}$ **Gradual Typing**

$$(S\mathrm{x})\frac{x : S \in \Gamma}{\Gamma \vdash x : S} \qquad (S\mathrm{n})\frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad (S\mathrm{b})\frac{}{\Gamma \vdash b : \mathsf{Bool}}$$

$$(S\mathrm{app})\frac{\left(\Gamma \vdash t_i : S_i\right)_{i \in \{1,2\}} \qquad S_2 \lesssim \widetilde{dom}(S_1)}{\Gamma \vdash t_1\ t_2 : \widetilde{cod}(S_1)} \qquad (S{+})\frac{\left(\Gamma \vdash t_i : S_i\right)_{i \in \{1,2\}} \qquad \left(S_i \lesssim \mathsf{Int}\right)_{i \in \{1,2\}}}{\Gamma \vdash t_1 + t_2 : \mathsf{Int}}$$

$$(S\mathrm{if})\frac{\left(\Gamma \vdash t_i : S_i\right)_{i \in \{1,2,3\}} \qquad S_1 \lesssim \mathsf{Bool}}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : S_2 \mathbin{\widetilde{\vee}} S_3} \qquad (S\mathrm{proj})\frac{\Gamma \vdash t : S}{\Gamma \vdash t.l : \widetilde{proj}(S, l)}$$

$$(S\lambda)\frac{\Gamma, x : S_1 \vdash t : S_2}{\Gamma \vdash (\lambda x : S_1.t) : S_1 \to S_2} \qquad (S{::})\frac{\Gamma \vdash t : S \qquad S \lesssim S_1}{\Gamma \vdash (t :: S_1) : S_1} \qquad (S\mathrm{rec})\frac{\left(\Gamma \vdash t_i : S_i\right)_{0 \le i \le n}}{\Gamma \vdash [\overline{l_i = t_i}] : [\overline{l_i : S_i}]}$$

Helper Functions

$$\begin{array}{ll}
\widetilde{dom} : \textsc{GType} \rightharpoonup \textsc{GType} & \widetilde{cod} : \textsc{GType} \rightharpoonup \textsc{GType} \\
\widetilde{dom}(S_1 \to S_2) = S_1 & \widetilde{cod}(S_1 \to S_2) = S_2 \\
\widetilde{dom}(?) = ? & \widetilde{cod}(?) = ? \\
\widetilde{dom}(S) \text{ undefined otherwise} & \widetilde{cod}(S) \text{ undefined otherwise}
\end{array}$$

$$\begin{array}{l}
\widetilde{proj} : \textsc{GType} \times \textsc{Label} \rightharpoonup \textsc{GType} \\
\widetilde{proj}([l : S, \overline{l_i : S_i}, *], l) = S \\
\widetilde{proj}([\overline{l_i : S_i}, ?], l) = ? \quad \text{if} \quad l \notin \{\, \overline{l_i} \,\} \\
\widetilde{proj}(?, l) = ? \\
\widetilde{proj}(S, l) \text{ undefined otherwise}
\end{array}$$

Figure 3.6: GTFL$_{\lesssim}$: Static Semantics, part 1/2

$\boxed{S \lesssim S}$ **Consistent Subtyping**

$$\frac{}{? \lesssim S} \qquad \frac{}{S \lesssim ?} \qquad \frac{}{\mathsf{Int} \lesssim \mathsf{Int}} \qquad \frac{}{\mathsf{Bool} \lesssim \mathsf{Bool}}$$

$$\frac{S_{21} \lesssim S_{11} \quad S_{12} \lesssim S_{22}}{S_{11} \to S_{12} \lesssim S_{21} \to S_{22}} \qquad \frac{\left(S_{i1} \lesssim S_{i2}\right)_{0 \le i \le n}}{[\overline{l_i : S_{i1}}, \overline{l_j : S_j}] \lesssim [\overline{l_i : S_{i2}}, *]}$$

$$\frac{\left(S_{i1} \lesssim S_{i2}\right)_{0 \le i \le n}}{[\overline{l_i : S_{i1}}, \overline{l_j : S_j}, ?] \lesssim [\overline{l_i : S_{i2}}, \overline{l_k : S_k}, *]}$$

$\boxed{S \sqsubseteq S}$ **Precision**

$$\frac{S_{11} \sqsubseteq S_{21} \quad S_{12} \sqsubseteq S_{22}}{S_{11} \to S_{12} \sqsubseteq S_{21} \to S_{22}} \qquad \frac{\left(S_{1i} \sqsubseteq S_{2i}\right)_{0 \le i \le n}}{[\overline{l_i : S_{1i}}] \sqsubseteq [\overline{l_i : S_{2i}}]} \qquad \frac{\left(S_{1i} \sqsubseteq S_{2i}\right)_{0 \le i \le n}}{[\overline{l_i : S_{1i}}, \overline{l_j : S_{1j}}, *] \sqsubseteq [\overline{l_i : S_{2i}}, ?]}$$

$$\frac{}{S \sqsubseteq ?} \qquad \frac{}{\mathsf{Int} \sqsubseteq \mathsf{Int}} \qquad \frac{}{\mathsf{Bool} \sqsubseteq \mathsf{Bool}}$$

Figure 3.7: GTFL$_{\lesssim}$: Static Semantics, Part 2/2

of it from the programmer: the abstraction always reflects all relevant instances of static subtyping.

3.2.2.1 *An example*

The following (somewhat contrived) example program demonstrates some of the features and intended capabilities of GTFL$_{\lesssim}$, especially the semantics of gradual rows. For succinctness, we assume let binding, which can be easily added to the language.

$$\begin{aligned}
&\mathsf{let\ sum} = \lambda(\mathsf{hasM} : \mathsf{Bool}).\, \lambda(\mathsf{x} : [\mathsf{f} : \mathsf{Int}, ?]).\\
&\qquad\qquad \mathsf{if\ hasM\ then\ x.f} + \mathsf{x.m\ else\ x.f} + \mathsf{x.q}\\
&\mathsf{in}\ (\mathsf{sum\ true}\ [\mathsf{f} = 6, \mathsf{m} = 2]) + (\mathsf{sum\ false}\ [\mathsf{f} = 6, \mathsf{q} = 2])
\end{aligned}$$

The sum function takes a record f, and a boolean value hasM, and uses hasM to determine which field to add to x.f. The gradual row type ascribed to x ensures statically that the record argument contains an f field of type Int, but makes no commitment regarding what other fields may be present. Thus the body of the function type checks despite referring to x.m and x.q.

The program successfully calculates the sum for two different records that have different fields. The hasM argument indicates which branch of execution is the right one. This small program demonstrates the possibility of at least partially statically checking the program, while deferring checks for extra fields to run time. We can extrapolate from this

program to a larger application that fruitfully exploits field dynamism, while statically checking stable record components.

This language design makes the possibility of dynamic checking evident in the types. The gradual type $[\mathsf{f} : \mathsf{Int}, ?]$ indicates that the record may have more fields, which might be used by the program.

Furthermore, the type structure of a program can be exploited to control the amount and scope of necessary dynamic checking. For instance, if the branches of the conditional made repeated use of extra fields, then each access would require a run-time check. However, we can ensure that these are checked statically (and centralize the type assumptions) using a type ascription. For example, we can replace the consequent branch $\mathsf{x.f} + \mathsf{x.m}$ with

$$\mathsf{let\ y} = (\mathsf{x} :: [\mathsf{f} : \mathsf{Int}, \mathsf{m} : \mathsf{Int}])\ \mathsf{in\ y.f} + \mathsf{y.m}$$

This change would move the initial field check (for $\mathsf{x.m}$) and field type check (that its type is Int) to the ascription: the body of the consequent let would be fully statically checked. This use of ascription is similar to downcasting in object-oriented languages, but is justified by the presence of the imprecise gradual row in the type of sum's argument.

Now for some bad news. Unfortunately, GTFL$_{\lesssim}$ as designed by Garcia, Clark, and Tanter (2016) does not enforce its type abstractions to the extent that one might expect. Sound gradual type regimes typically have the property that precise type information is persistent. For instance, if we replace the record $[\mathsf{f} = 6, \mathsf{m} = 2]$ in the example with $([\mathsf{f} = 6, \mathsf{m} = 2] :: [\mathsf{f} : \mathsf{Int}])$, then one would expect the program to type check, but fail at run time because of the stricter type constraint we introduce in the cast, which should stop the field m from being visible. In this way, the language would guarantee that the *m* field's very existence is encapsulated: no client would be able to violate that guarantee.

To our surprise, however, GTFL$_{\lesssim}$ executes this program successfully, for essentially the same reason that the errant example at the beginning of the chapter succeeds. We consider this a flaw in the GTFL$_{\lesssim}$ semantics: type abstractions should be respected. Bear in mind that a language designer may desire a gradual language that treats all gradual record types as gradual rows, thereby baking downcasting into the semantics. This can easily be supported as syntactic sugar over the GTFL$_{\lesssim}$ surface syntax, rather than as an accident of AGT-induced semantics. In short, the semantics of an AGT-induced language should precisely enforce types. In the following, we diagnose this failure of type abstraction and show how to properly ensure this property with a refinement to the AGT framework.

3.2.3 *Runtime Language*

Since a gradual language defers some checks to run time, its dynamic semantics must account for those checks. Using the approach discussed in Section 3.1.7 for AGTLC and inspired by Toro, Garcia, and Tanter (2018), we capture GTFL$_{\lesssim}$ checks in an internal language that decorates programs with run-time type information.

3.2.3.1 *Syntax and Static Semantics*

Figure 3.8 presents the syntax and type system of the GTFL$_{\lesssim}$ Runtime Language (RL). The justification for this language is similar to that of ARL with respect to AGTLC.

Syntax

$$
\begin{array}{rcll}
\varepsilon \in \text{Ev} & = & \{\, \langle S_1, S_2 \rangle \mid \vdash \langle S_1, S_2 \rangle \ \mathbf{wf} \,\} & \text{(evidence objects)} \\
\mathbf{e} \in \text{RTerm} & ::= & \mathbf{n} \mid \mathbf{b} \mid \mathbf{x} \mid \lambda \mathbf{x}.\mathbf{e} \mid \varepsilon\mathbf{e}\ \varepsilon\mathbf{e} \mid \varepsilon\mathbf{e} + \varepsilon\mathbf{e} \mid & \\
 & & \text{if } \varepsilon\mathbf{e} \text{ then } \varepsilon\mathbf{e} \text{ else } \varepsilon\mathbf{e} \mid [\overline{\mathbf{l} = \mathbf{e}}] \mid \varepsilon\mathbf{e}.l \mid \varepsilon\mathbf{e} & \text{(runtime terms)} \\
\mathbf{u} \in \text{RawValue} & ::= & \mathbf{n} \mid \mathbf{b} \mid \mathbf{x} \mid \lambda \mathbf{x}.\mathbf{e} \mid [\overline{\mathbf{l} = \mathbf{v}}] & \text{(raw values)} \\
\mathbf{v} \in \text{Value} & ::= & \mathbf{u} \mid \varepsilon\mathbf{u} & \text{(values)}
\end{array}
$$

$\boxed{\mathbf{e} \blacktriangleright \Gamma \vdash t : S}$ **Runtime Typing**

$$
(S\text{n})\frac{}{\mathbf{n} \blacktriangleright \Gamma \vdash n : \mathsf{Int}} \qquad (S\text{b})\frac{}{\mathbf{b} \blacktriangleright \Gamma \vdash b : \mathsf{Bool}} \qquad (S\text{x})\frac{x : S' \in \Gamma}{\mathbf{x} \blacktriangleright \Gamma \vdash x : S'}
$$

$$
(S\text{app})\frac{\left(\mathbf{e}_i \blacktriangleright \Gamma \vdash t_i : S_i\right)_{i \in \{1,2\}} \qquad \varepsilon_1 \vdash S_1 \lesssim S_1' \to S_2' \qquad \varepsilon_2 \vdash S_2 \lesssim S_1'}{(\varepsilon_1\mathbf{e}_1\ \varepsilon_2\mathbf{e}_2) \blacktriangleright \Gamma \vdash t_1\ t_2 : S_2'}
$$

$$
(S\text{if})\frac{\left(\mathbf{e}_i \blacktriangleright \Gamma \vdash t_i : S_i\right)_{i \in \{1,2,3\}} \qquad \varepsilon_1 \vdash S_1 \lesssim \mathsf{Bool} \qquad \varepsilon_2 \vdash S_2 \lesssim S_2 \mathbin{\widetilde{\vee}} S_3 \qquad \varepsilon_3 \vdash S_3 \lesssim S_2 \mathbin{\widetilde{\vee}} S_3}{(\text{if } \varepsilon_1\mathbf{e}_1 \text{ then } \varepsilon_2\mathbf{e}_2 \text{ else } \varepsilon_3\mathbf{e}_3) \blacktriangleright \Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : S_2 \mathbin{\widetilde{\vee}} S_3}
$$

$$
(S\text{proj})\frac{\mathbf{e} \blacktriangleright \Gamma \vdash t : S \qquad \varepsilon \vdash S \lesssim [l : S']}{\varepsilon\mathbf{e}.l \blacktriangleright \Gamma \vdash t.l : S'} \qquad (S\text{rec})\frac{\left(\mathbf{e}_i \blacktriangleright \Gamma \vdash t_i : S_i\right)_{0 \le i \le n}}{[\overline{\mathbf{l}_i = \mathbf{e}_i}] \blacktriangleright \Gamma \vdash [\overline{l_i = t_i}] : [\overline{l_i : S_i}]}
$$

$$
(S\lambda)\frac{\mathbf{e} \blacktriangleright \Gamma, x : S_1 \vdash t : S_2}{(\lambda x.\mathbf{e}) \blacktriangleright \Gamma \vdash (\lambda x : S_1.t) : S_1 \to S_2}
$$

$$
(S{::})\frac{\mathbf{e} \blacktriangleright \Gamma \vdash t : S \qquad \varepsilon \vdash S \lesssim S'}{\varepsilon\mathbf{e} \blacktriangleright \Gamma \vdash (t :: S') : S'} \qquad (S{+})\frac{\left(\mathbf{e}_i \blacktriangleright \Gamma \vdash t_i : S_i\right)_{i \in \{1,2\}} \qquad \left(\varepsilon_i \vdash S_i \lesssim \mathsf{Int}\right)_{i \in \{1,2\}}}{(\varepsilon_1\mathbf{e}_1 + \varepsilon_2\mathbf{e}_2) \blacktriangleright \Gamma \vdash t_1 + t_2 : \mathsf{Int}}
$$

$\boxed{\varepsilon \vdash S_1 \lesssim S_2}$ **Evidence for Consistent Subtyping**

$$
\frac{\vdash \langle S_1', S_2' \rangle \ \mathbf{wf} \qquad S_1' \sqsubseteq S_1 \qquad S_2' \sqsubseteq S_2}{\langle S_1', S_2' \rangle \vdash S_1 \lesssim S_2}
$$

$\boxed{\vdash \varepsilon\ \mathbf{wf}}$ **Well-formed Evidence**

$$
\frac{S \in \{\, \mathsf{Int}, \mathsf{Bool}, ? \,\}}{\vdash \langle S, S \rangle \ \mathbf{wf}} \qquad \frac{\vdash \langle S_{21}, S_{11} \rangle \ \mathbf{wf} \qquad \vdash \langle S_{12}, S_{22} \rangle \ \mathbf{wf}}{\vdash \langle S_{11} \to S_{12}, S_{21} \to S_{22} \rangle \ \mathbf{wf}}
$$

$$
\frac{\left(\vdash \langle S_{i1}, S_{i2} \rangle \ \mathbf{wf}\right)_{0 \le i \le n} \qquad \langle *_1, *_2 \rangle \neq \langle \cdot, ? \rangle}{\vdash \langle [\overline{l_i : S_{i1}}, *_1], [\overline{l_i : S_{i2}}, *_2] \rangle \ \mathbf{wf}} \qquad \frac{\left(\vdash \langle S_{i1}, S_{i2} \rangle \ \mathbf{wf}\right)_{0 \le i \le n}}{\vdash \langle [\overline{l_i : S_{i1}}, \overline{l_j : S_j}^{+}, *_1], [\overline{l_i : S_{i2}}, *_2] \rangle \ \mathbf{wf}}
$$

Figure 3.8: GTFL$_{\lesssim}$: Runtime Language Static Semantics

Naturally, the structure of RL typing rules closely mirrors those of GTFL$_{\lesssim}$, but differs on a few details. First, the (Sx) and (S::) rules show how missing source type information appears as part of the corresponding source derivation only, and not as part of the term. Gradual types highlighted in grey in Figure 3.8 are artifacts of source type-checking that cannot be recovered by examining the runtime term itself. We exploit this information only to establish the type safety of RL programs: the runtime never needs to reconstruct a typing derivation from a naked RL term, they are artifacts of our proofs. Second, each instance of consistent subtyping from GTFL$_{\lesssim}$ is now decorated with an evidence object that supports the judgment. For instance, the RL term $\varepsilon\mathbf{e}$ corresponds via typing directly to an ascription expression $t :: S_0$. The *evidence judgment* $\varepsilon \vdash S \lesssim S_0$ says that ε confirms that $S \lesssim S_0$ is plausible.

The (Sif) and (Sapp) rules consider evidence for extra consistent subtyping judgments that were only implicitly required by the GTFL$_{\lesssim}$ rules. For instance, (Sapp) demands that $S_1 \lesssim \widetilde{dom}(S_1) \to \widetilde{cod}(S_1)$. This extra constraint is implied by the GTFL$_{\lesssim}$ requirement that $\widetilde{dom}(S_1)$ and $\widetilde{cod}(S_1)$, an argument and a result type, respectively, are well defined. The elaboration rule, and the corresponding RL term, make this implicit constraint explicit so as to enforce type structure at run time. Similarly, the (Sif) type $S_2 \tilde{\vee} S_3$ imposes implicit consistent subtyping constraints on each branch of the conditional.

In RL, an evidence object is a pair of gradual types that are tightly related to one another, and also related to the consistent subtyping judgment that they support. The well-formedness judgment $\vdash \varepsilon$ **wf** imposes the same invariants on evidence we described for AGTLC, but for the subtyping judgment.

Proposition 15. *The judgment stating that an evidence object is well-formed,* $\vdash \langle S_1, S_2 \rangle$ **wf**, *is equivalent to the combination of the following criteria:*

1. *The gradual types in the evidence object are related by consistent* subtyping *(*$S_1 \lesssim S_2$, i.e. $T_1 <: T_2$ *for some* $T_1 \sqsubseteq S_1$ *and* $T_2 \sqsubseteq S_2$*).*

2. *The gradual types in the evidence object are the least upper bound over precision among all the static types related by* subtyping*. Formally, suppose* $S_1', S_2' \in$ GType *and whenever* $T_1 <: T_2$, $T_1 \sqsubseteq S_1$, *and* $T_2 \sqsubseteq S_2$, *we also have* $T_1 \sqsubseteq S_1'$ *and* $T_2 \sqsubseteq S_2'$. *Then* $S_1 \sqsubseteq S_1'$ *and* $S_2 \sqsubseteq S_2'$.

3.2.3.2 *Dynamic Semantics*

Figure 3.9 presents a reduction semantics for RL. The notions of reduction augment standard reduction steps with operations that manipulate evidence. We first consider these reductions at a high-level, and then delve into the role that evidence plays in these reductions. Addition ignores its associated evidence and behaves as usual. The intuition behind this is that the evidence is now superfluous: $\mathbf{n}_1$ and $\mathbf{n}_2$ evidently have type Int, because they are integers. Similarly, conditionals ignore the evidence associated with the predicate, because it is evidently a boolean value. The chosen branch's evidence is propagated as-is to enforce its type invariants. Record projection selects the relevant field of a record but also applies the *iproj* inversion operator to the evidence that is associated to the projection operator, in order to extract evidence that is relevant to the projected value. Function application is the most complex rule, but it follows the same structure as

Syntax

$$
\begin{array}{rcll}
E \in \text{ECtxt} & ::= & \square \mid E[[\overline{l = \mathbf{v}}, l = \square, \overline{l = \mathbf{e}}]] \mid E[F[\varepsilon\square]] & \text{(evaluation contexts)} \\
F \in \text{EvFrame} & ::= & \square \mid \square + \varepsilon\mathbf{e} \mid \varepsilon\mathbf{u} + \square \mid \square\ \varepsilon\mathbf{e} \mid \varepsilon\mathbf{u}\ \square \mid \square.l & \\
& \mid & \mathsf{if}\ \square\ \mathsf{then}\ \varepsilon\mathbf{e}\ \mathsf{else}\ \varepsilon\mathbf{e} & \text{(evidence frame)}
\end{array}
$$

$\boxed{\mathbf{e} \rightsquigarrow \mathbf{e},\ \mathbf{e} \rightsquigarrow \mathbf{error}}$ **Notions of Reduction**

$$
\begin{array}{rll}
\varepsilon_1\mathbf{n}_1 + \varepsilon_2\mathbf{n}_2 \rightsquigarrow \mathbf{n}_3 & & \mathbf{n}_3 = \mathbf{n}_1 + \mathbf{n}_2 \\
\varepsilon_1(\lambda\mathbf{x}.\mathbf{e})\ \varepsilon_2\mathbf{u} \rightsquigarrow \begin{cases} \big(\mathit{icod}(\varepsilon_1)\big)\big([(\varepsilon_3\mathbf{u})/\mathbf{x}]\mathbf{e}\big) & \\ \mathbf{error} \quad \text{if } \varepsilon_3 \text{ not defined} \end{cases} & & \varepsilon_3 = (\varepsilon_2 \fatsemi \mathit{idom}(\varepsilon_1)) \\
\mathsf{if}\ \varepsilon_1\mathsf{true}\ \mathsf{then}\ \varepsilon_2\mathbf{e}_2\ \mathsf{else}\ \varepsilon_3\mathbf{e}_3 \rightsquigarrow \varepsilon_2\mathbf{e}_2 & & \\
\mathsf{if}\ \varepsilon_1\mathsf{false}\ \mathsf{then}\ \varepsilon_2\mathbf{e}_2\ \mathsf{else}\ \varepsilon_3\mathbf{e}_3 \rightsquigarrow \varepsilon_3\mathbf{e}_3 & & \\
\varepsilon[\overline{l_i = \mathbf{v}_i}].l_j \rightsquigarrow \mathit{iproj}(\varepsilon, l_j)\mathbf{v}_j & &
\end{array}
$$

$\boxed{\mathbf{e} \longrightarrow \mathbf{e},\ \mathbf{e} \longrightarrow \mathbf{error}}$ **Contextual Reduction**

$$
\frac{\mathbf{e} \rightsquigarrow \mathbf{e}'}{E[\mathbf{e}] \longrightarrow E[\mathbf{e}']} \qquad \frac{\mathbf{e} \rightsquigarrow \mathbf{error}}{E[\mathbf{e}] \longrightarrow \mathbf{error}}
$$

$$
\frac{\varepsilon_2 \fatsemi \varepsilon_1 \text{ not defined}}{E[F[\varepsilon_1\varepsilon_2\mathbf{u}]] \longrightarrow \mathbf{error}} \qquad \frac{}{E[F[\varepsilon_1\varepsilon_2\mathbf{u}]] \longrightarrow E[F[(\varepsilon_2 \fatsemi \varepsilon_1)\mathbf{u}]]}
$$

Helper Functions

$$
\begin{array}{l}
\textsc{FunEv} = \{\, \varepsilon \in \textsc{Ev} \mid \varepsilon \vdash S_{11} \to S_{12} \lesssim S_{21} \to S_{22} \,\} \\
\textsc{IProjDomain} = \{\, \langle \varepsilon, l \rangle \in \textsc{Ev} \times \textsc{Label} \mid \varepsilon \vdash [l : S, \overline{l_i : S_i}, *_1] \lesssim [l : S', \overline{l_j : S_j}, *_2] \,\}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{idom} : \textsc{FunEv} \to \textsc{Ev} & \mathit{icod} : \textsc{FunEv} \to \textsc{Ev} \\
\mathit{idom}(S_1, S_2) = \langle \widetilde{\mathit{dom}}(S_2), \widetilde{\mathit{dom}}(S_1) \rangle & \mathit{icod}(S_1, S_2) = \langle \widetilde{\mathit{cod}}(S_1), \widetilde{\mathit{cod}}(S_2) \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathit{iproj} : \textsc{IProjDomain} \to \textsc{Ev} \\
\mathit{iproj}(\langle S_1, S_2 \rangle, l) = \langle \widetilde{\mathit{proj}}(S_1, l), \widetilde{\mathit{proj}}(S_2, l) \rangle
\end{array}
$$

Figure 3.9: GTFL$_{\lesssim}$: Runtime Language Dynamic Semantics

in AGTLC. Using the *idom* and *icod* inversion operators, it extracts evidence associated with the domain and codomain of the function subterm, composes the domain evidence with the argument evidence, and then associates the codomain evidence with the eventual result of the call. If composition fails, then the entire application fails.

To understand the above behaviours, it helps again to think of the operators like *iproj* as run-time representations of *inversion principles* for consistent subtyping. For instance, given the inductive definition of consistent subtyping in Figure 3.6, we can prove the following inversion lemma.

Proposition 16. *Consider two record types related by consistent subtyping. For every field in the super type, the field is therefore also present in the subtype, and the mapped types are related by consistent subtyping. Formally, if $l_k \in \{\overline{l_j}\}$ and $[\overline{l_i : S_i}] \lesssim [\overline{l_j : S'_j}]$ then $S_k \lesssim S'_k$.*

The *iproj* operator reifies this inversion principle (and those for gradual rows) at run time: If $\varepsilon \vdash [\overline{l_i : S_i}] \lesssim [\overline{l_j : S'_j}]$ then $iproj(\varepsilon, l_k) \vdash S_k \lesssim S'_k$. So just as inversion principles are used to prove static type safety, evidence inversion operators are used to enforce type invariants at run time.

While evidence inversion operations are used to simulate inversion principles, evidence composition $\varepsilon_1 \fatsemi \varepsilon_2$ is used to monitor transitivity of consistent subtyping, much like in AGTLC.

The defining correctness criteria for consistent transitivity are as follows.[13]

Definition 11 (Candidates for transitivity)**.** *Suppose two evidence judgments that share a supported gradual type "in the middle" (the supported super type in one judgment is the same as the supported sub type in the other), formally $\varepsilon_1 = \langle S'_1, S'_{21} \rangle \vdash S_1 \lesssim S_2$ and $\varepsilon_2 = \langle S'_{22}, S'_3 \rangle \vdash S_2 \lesssim S_3$.*

1. *We then call (ε_1,ε_2)* plausibly transitive *if we can form a transitive chain of three static types related by subtyping, each less imprecise than the type in the appropriate portion of the evidence objects, while reusing the same type in the middle for the portions of both evidence objects that were also "in the middle". Formally, $T_1 \sqsubseteq S'_1$ and $S'_{21} \sqsupseteq T_2 \sqsubseteq S'_{22}$, and $T_3 \sqsubseteq S'_3$ hold for some triple $T_1 <: T_2 <: T_3$;*

2. *We then call $\varepsilon' = \langle S''_1, S''_3 \rangle$ a* candidate for transitivity of (ε_1,ε_2) *if (ε_1,ε_2) is plausibly transitive, and if for every triple of static types that could be used to justify (ε_1,ε_2) as plausibly transitive, the types on the extremes are also bound via precision to the respective type in ε'. Formally, every triple $T_1 <: T_2 <: T_3$ such that $T_1 \sqsubseteq S'_1$ and $S'_{21} \sqsupseteq T_2 \sqsubseteq S'_{22}$, and $T_3 \sqsubseteq S'_3$ implies $T_1 \sqsubseteq S''_1$ and $T_3 \sqsubseteq S''_3$;*

Proposition 17. *Suppose two evidence judgments that share a supported gradual type "in the middle", formally $\varepsilon_1 = \langle S'_1, S'_{21} \rangle \vdash S_1 \lesssim S_2$ and $\varepsilon_2 = \langle S'_{22}, S'_3 \rangle \vdash S_2 \lesssim S_3$. Then ($\varepsilon_1$,$\varepsilon_2$) has a candidate for transitivity if and only if it has a least candidate over the precision ($\sqsubseteq$) order, denoted $\varepsilon_1 \fatsemi \varepsilon_2$. Furthermore, this least candidate also supports consistent subtyping among the types on the extremes, formally $(\varepsilon_1 \fatsemi \varepsilon_2) \vdash S_1 \lesssim S_3$.*

Contextual reduction formalizes three kinds of reduction. First, it captures how notions of reduction apply in evaluation position. Second, it captures how an error during a notion of reduction aborts the entire computation. Third, it reflects how sequences of evidence objects are composed, producing new evidence on success or signalling a run-time type error on failure.

3.2.3.3 *Type Safety*

Like ARL for AGTLC, RL satisfies standard type safety (Wright and Felleisen, 1994): the semantics explicitly categorizes all well-typed terms as reducible or as canonical forms. By appropriate progress and preservation theorems, programs in RL do not get stuck, but they may signal run-time type errors.

Unlike typical preservation theorems for languages with subtyping (e.g. Benjamin C. Pierce (2002)), the type of the resulting term remains *exactly* the same as the source term. This is critical because consistent subtyping *does not* denote safe substitutability.[14] Safety requires that any use of consistent subtyping must be mediated by run-time evidence. Even if a subterm does evolve to a consistent subtype, it is wrapped with run-time evidence that explicitly coerces it back to the original consistent supertype.

3.2.4 *Elaboration*

GTFL$_{\lesssim}$ source programs are elaborated to RL programs by type-directed translation (Figure 3.10). Like for AGTLC, the elaboration process is quite uniform and exhibits the tight connection between GTFL$_{\lesssim}$ and RL.

Similarly to the elaboration of AGTLC, each elaboration rule for GTFL$_{\lesssim}$ corresponds directly to a source GTFL$_{\lesssim}$ typing rule. In particular, each premise typing judgment becomes a corresponding elaboration judgment, and each consistent subtyping judgment introduces an evidence object, using the initial evidence operator,[15] now for consistent subtyping instead of the consistent equality we used in AGTLC.

The ($\rightsquigarrow$if) and ($\rightsquigarrow$app) elimination rules produce evidence for extra consistent subtyping judgments that were not evident in the corresponding (Sif) and (Sapp) GTFL$_{\lesssim}$ rules. For instance, ($\rightsquigarrow$app) demands that $S_1 \lesssim \widetilde{dom}(S_1) \to \widetilde{cod}(S_1)$. This extra constraint was implied by the GTFL$_{\lesssim}$ requirement that $\widetilde{cod}(S_1)$ be well-defined. The elaboration rule and the corresponding RL term make this implicit constraint explicit, because it must be enforced at run time. Similarly, the ($\rightsquigarrow$if) type $S_2 \mathbin{\widetilde{\vee}} S_3$ imposes implicit consistent subtyping constraints on each branch of the conditional.

The translation for GTFL$_{\lesssim}$ also preserves well-formedness and has the replicant property.

Proposition 18 (Translation preserves well-formedness)**.** *The runtime term resulting from elaboration represents the typing derivation that guided the elaboration process. Formally, if* $\Gamma \vdash t \rightsquigarrow \mathbf{e} : S$ *then* $\mathbf{e} \blacktriangleright \Gamma \vdash t : S$.

Proposition 19 (Replicant)**.** *If* $\mathbf{e} \blacktriangleright \Gamma \vdash t : S$ *and* $\Gamma \vdash t \rightsquigarrow \mathbf{e}' : S$ *then* $\mathbf{e} \sqsubseteq \mathbf{e}'$.

$\boxed{\Gamma \vdash t \rightsquigarrow \mathbf{e} : S}$ **Elaboration**

$$(\rightsquigarrow\text{x})\frac{x : S \in \Gamma}{\Gamma \vdash x \rightsquigarrow \mathbf{x} : S} \qquad (\rightsquigarrow\text{n})\frac{}{\Gamma \vdash n \rightsquigarrow \mathbf{n} : \mathsf{Int}} \qquad (\rightsquigarrow\text{b})\frac{}{\Gamma \vdash b \rightsquigarrow \mathbf{b} : \mathsf{Bool}}$$

$$(\rightsquigarrow\text{app})\frac{\begin{array}{cc}\Gamma \vdash t_1 \rightsquigarrow \mathbf{e}_1 : S_1 & \varepsilon_1 = \mathcal{I}[\![S_1 \lesssim \widetilde{dom}(S_1) \to \widetilde{cod}(S_1)]\!] \\ \Gamma \vdash t_2 \rightsquigarrow \mathbf{e}_2 : S_2 & \varepsilon_2 = \mathcal{I}[\![S_2 \lesssim \widetilde{dom}(S_1)]\!]\end{array}}{\Gamma \vdash t_1\ t_2 \rightsquigarrow \varepsilon_1\mathbf{e}_1\ \varepsilon_2\mathbf{e}_2 : \widetilde{cod}(S_1)}$$

$$(\rightsquigarrow+)\frac{\Big(\Gamma \vdash t_i \rightsquigarrow \mathbf{e}_i : S_i\Big)_{i\in\{1,2\}} \quad \Big(\varepsilon_i = \mathcal{I}[\![S_i \lesssim \mathsf{Int}]\!]\Big)_{i\in\{1,2\}}}{\Gamma \vdash t_1 + t_2 \rightsquigarrow \varepsilon_1\mathbf{e}_1 + \varepsilon_2\mathbf{e}_2 : \mathsf{Int}}$$

$$(\rightsquigarrow\text{if})\frac{\begin{array}{c}\Big(\Gamma \vdash t_i \rightsquigarrow \mathbf{e}_i : S_i\Big)_{i\in\{1,2,3\}} \quad \Big(\varepsilon_i = \mathcal{I}[\![S_i \lesssim S_2 \mathbin{\widetilde{\vee}} S_3]\!]\Big)_{i\in\{2,3\}} \\ \varepsilon_1 = \mathcal{I}[\![S_1 \lesssim \mathsf{Bool}]\!]\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \rightsquigarrow \\ \qquad \mathsf{if}\ \varepsilon_1\mathbf{e}_1\ \mathsf{then}\ \varepsilon_2\mathbf{e}_2\ \mathsf{else}\ \varepsilon_3\mathbf{e}_3 : S_2 \mathbin{\widetilde{\vee}} S_3\end{array}}$$

$$(\rightsquigarrow\text{proj})\frac{\Gamma \vdash t \rightsquigarrow \mathbf{e} : S \quad \varepsilon = \mathcal{I}[\![S \lesssim [l : \widetilde{proj}(S, l)]]\!]}{\Gamma \vdash t.l \rightsquigarrow \varepsilon\mathbf{e}.l : \widetilde{proj}(S, l)}$$

$$(\rightsquigarrow\lambda)\frac{\Gamma, x : S_1 \vdash t \rightsquigarrow \mathbf{e} : S_2}{\Gamma \vdash (\lambda x : S_1.t) \rightsquigarrow (\lambda\mathbf{x}.\mathbf{e}) : S_1 \to S_2} \qquad (\rightsquigarrow::)\frac{\Gamma \vdash t \rightsquigarrow \mathbf{e} : S \quad \varepsilon = \mathcal{I}[\![S \lesssim S_1]\!]}{\Gamma \vdash (t :: S_1) \rightsquigarrow \varepsilon\mathbf{e} : S_1}$$

$$(\rightsquigarrow\text{rec})\frac{\Big(\Gamma \vdash t_i \rightsquigarrow \mathbf{e}_i : S_i\Big)_{0\le i\le n}}{\Gamma \vdash [\overline{l_i = t_i}] \rightsquigarrow [\overline{l_i = \mathbf{e}_i}] : [\overline{l_i : S_i}]}$$

Figure 3.10: GTFL$_{\lesssim}$: Elaboration

3.2.5 *GTFL$_{\lesssim}$ Shortcomings Revisited*

Using the semantics of GTFL$_{\lesssim}$ presented above, we can more closely examine the program from the beginning of the chapter that failed to protect type invariant boundaries. Our example program:

$$\begin{array}{l}\mathsf{let\ q : [x : Int] = [x = 5, y = true]\ in}\\ \quad\mathsf{let\ z : ? = q\ in}\\ \quad\quad\mathsf{z.y}\end{array}$$

elaborates to the following: [16]

$$\begin{array}{l}\mathsf{let\ q : [x : Int] = }\varepsilon_1\mathsf{[x = 5, y = true]\ in}\\ \quad\mathsf{let\ z : ? = }\varepsilon_2\mathsf{q\ in}\\ \quad\quad\varepsilon_3\mathsf{z.y}\end{array}$$

where

$$\begin{array}{lll}\varepsilon_1 & = \mathcal{I}[\![\mathsf{[x : Int, y : Bool]} \lesssim \mathsf{[x : Int]}]\!] & = \langle\mathsf{[x : Int, y : Bool], [x : Int]}\rangle\\ \varepsilon_2 & = \mathcal{I}[\![\mathsf{[x : Int]} \lesssim \mathsf{?}]\!] & = \langle\mathsf{[x : Int], [?]}\rangle\\ \varepsilon_3 & = \mathcal{I}[\![\mathsf{?} \lesssim \mathsf{[y : Bool]}]\!] & = \langle\mathsf{[y : Bool, ?], [y : Bool]}\rangle\end{array}$$

Evaluation proceeds as follows:

$$\begin{array}{l}\quad\mathsf{let\ q : [x : Int] = }\varepsilon_1\mathsf{[x = 5, y = true]\ in\ let\ z : ? = }\varepsilon_2\mathsf{q\ in\ }\varepsilon_3\mathsf{z.y}\\ \longrightarrow \mathsf{let\ z : ? = }\varepsilon_2\varepsilon_1\mathsf{[x = 5, y = true]\ in\ }\varepsilon_3\mathsf{z.y}\\ \longrightarrow \varepsilon_3\varepsilon_2\varepsilon_1\mathsf{[x = 5, y = true].y}\\ \longrightarrow (\varepsilon_3(\varepsilon_1 \fatsemi \varepsilon_2))\mathsf{[x = 5, y = true]).y}\\ \longrightarrow (((\varepsilon_1 \fatsemi \varepsilon_2) \fatsemi \varepsilon_3)\mathsf{[x = 5, y = true]).y}\\ \longrightarrow \varepsilon_4\mathsf{true}\end{array}$$

where

$$\begin{array}{rl}(\varepsilon_1 \fatsemi \varepsilon_2) &= \langle\mathsf{[x : Int, y : Bool], [?]}\rangle\\ ((\varepsilon_1 \fatsemi \varepsilon_2) \fatsemi \varepsilon_3) &= \langle\mathsf{[x : Int, y : Bool], [y : Bool]}\rangle\\ \varepsilon_4 &= \langle\mathsf{Bool, Bool}\rangle\end{array}$$

This program should signal a run-time type error, but it does not. Tracing its evaluation reveals what went wrong. First, observe that $\varepsilon_2 \fatsemi \varepsilon_3$ is undefined, even though $((\varepsilon_1 \fatsemi \varepsilon_2) \fatsemi \varepsilon_3)$ *is* defined. Now consider $(\varepsilon_1 \fatsemi \varepsilon_2) = \langle\mathsf{[x : Int, y : Bool], [?]}\rangle$ more closely. Appealing to our notion of well-formed evidence, this object contains –via precision– every valid static subtyping pair $\mathsf{[x : Int, y : Bool]} <: T$, including $T = \mathsf{[x : Int, y : Bool]}$.

However, if we consider the correctness criteria for Proposition 17, as applied to composing ε_1 with ε_2, then the only static types that complete the relevant triples $T_1 <: T_2 <: T_3$ are $T_3 = [\mathsf{x} : \mathsf{Int}]$ and $T_3 = [\,]$. Thus $[\mathsf{x} : \mathsf{Int}, \mathsf{y} : \mathsf{Bool}]$ is a spurious potential supertype due solely to the choice of evidence abstraction. These phenomena suggest that our abstraction for evidence is insufficiently precise. We fix this problem in Section 3.4.

3.2.6 Should *we be precise?*

Reasonable programmers may disagree with me about whether this example program should produce an error or not. After all, the value mapped by the identifier z *does* have a field y, and many languages allow for safe "downcasts" (type ascriptions where the declared type can be a subtype of the type of the ascribed expression). In that context, this program could successfully run to a value. However, this was not the case for the original statically-typed language to which we applied AGT. STFL$_{<:}$ did not allow for "downcasts". Only "upcasts" are accepted by the (T::) typing rule in Figure 3.5.

If we did not produce an error in this situation, typed-based reasoning in our gradual language would depart from the type-based reasoning expected in the original statically-typed language. We want types that allow for static and compositional reasoning about the behaviour of the program, as argued by Toro, Garcia, and Tanter (2018, page 16:3) in the more subtle context of security typing. Static reasoning means that we can expect values at runtime to behave as their static types declare them to behave. Compositional reasoning means that to reason about these values we are allowed to rely only on local type information, without needing to follow the global flow of a program to find the source of a particular value. To allow for this static and compositional reasoning about programs, the type constraints appended to a value as it flows through a program should never be discarded. Forward completeness guarantees that an AGT language does not discard any type constraints, and aborts execution when inconsistent type constraints are revealed.

Although I do not formally define this additive property of type constraints in this dissertation, future work should explore the connection between forward complete AGT and the notion of *type vigilance* introduced by Gierczak et al. (2024). Type vigilance provides a framework to justify whether the run-time of a gradual system enforces the guarantees of its static counterpart and, when it doesn't, can be used to obtain a *different* (and weaker) static type system that is the one actually being enforced by the run-time. It would not be surprising if precise AGT provides a design recipe for type-vigilant gradual languages, but I leave the formal connection between these two to future work.

3.3 UPDATING AGT: EVIDENCE GETS ITS OWN ABSTRACTION

Since our goal is to develop a general solution to the issues we show for GTFL$_{\lesssim}$, we ground our approach in terms of the AGT framework itself (Garcia, Clark, and Tanter, 2016). Our new approach improves on the original in that it clarifies the nature of *two* distinct but related abstractions: one for the static semantics and one for the dynamic semantics. This refinement is critical to our contributions.

The concept of evidence for consistent subtyping arises through an analogous process to the definition of gradual helper functions. Garcia, Clark, and Tanter (2016) define evidence by abstracting tuples of static types, filtering them *post-hoc*. Our approach refines theirs in a small but fundamental way. We introduce a *second* Galois connection, but this one is directly between consistent subtyping, regarded as a set of pairs of gradual types $\langle S_1, S_2 \rangle \in \lesssim$, and nonempty subsets of the static subtyping relation, *i.e.* sets of pairs $\mathcal{R} \subseteq <:$.

The key difference in this new approach is that the set of evidence objects form *in their entirety* an abstract interpretation of static subtyping. In contrast, the evidence objects introduced by (Garcia, Clark, and Tanter, 2016) were arbitrary pairs of gradual types. The concretization of an evidence object could therefore contain pairs of types not related by subtyping. Integrating subtyping into the conception of abstraction, and viewing composition as relational composition specifically for fragments of subtyping is critical to formulating the possibility, let alone the significance, of forward completeness. We introduce forward completeness in detail in Section 3.5.

We begin with concretization, which we refine to filter only types related by subtyping.

$$\gamma^{<:} : \lesssim \to \mathcal{P}^+(<:)$$
$$\gamma^{<:}(S_1, S_2) = \{\, \langle T_1, T_2 \rangle \mid T_1 \in \gamma(S_1), T_2 \in \gamma(S_2), \text{ and } T_1 <: T_2 \,\}\,.$$

The corresponding notion of abstraction is defined as follows

$$\alpha^{<:} : \mathcal{P}^+(<:) \to \lesssim$$
$$\alpha^{<:}(\mathcal{R}) = \langle \alpha(\pi_1(\mathcal{R})), \alpha(\pi_2(\mathcal{R})) \rangle\,.$$

Abstraction appeals to two point-wise projection functions that operate on sets of pairs. These recur in later developments.

$$\pi_1, \pi_2 : \mathcal{P}(<:) \to \mathcal{P}(\textsc{Type})$$
$$\pi_i(\{\, \overline{T_1 <: T_2} \,\}) = \{\, \overline{T_i} \,\}$$

The functions $\gamma^{<:}$ and $\alpha^{<:}$ form a Galois connection, but *they do not form a Galois insertion*. For example: $\alpha^{<:}(\gamma^{<:}(\mathsf{Int}, ?)) = \langle \mathsf{Int}, \mathsf{Int} \rangle$. In fact, $\alpha^{<:}(\gamma^{<:}(S_1, S_2))$ is equivalent, by design, to the initial evidence function $\mathcal{I}[\![S_1 \lesssim S_2]\!]$ from Figure 3.10. The $\mathcal{I}$ function simply extends its domain to accept, but be undefined for, inconsistent gradual types $S_1 \not\lesssim S_2$. Thus, evidence objects ε are exactly those pairs of gradual types in the image of $\alpha^{<:} \circ \gamma^{<:}$.

Furthermore, composing evidence objects using the consistent transitivity operator can be defined in terms of *relational composition* of subsets of static subtyping $<:$. Read the

following operation as "compose all subtyping pairs from the respective concretizations, and then abstract back the set into a new evidence object":

$$\varepsilon_1 \fatsemi \varepsilon_2 = \alpha^{<:}(Id^{+}(\gamma^{<:}(\varepsilon_1)\,;\gamma^{<:}(\varepsilon_2)))$$
$$\text{where } \mathcal{R}_1\,;\mathcal{R}_2 = \{\,\langle T_1, T_3\rangle \mid T_1\ \mathcal{R}_1\ T_2 \text{ and } T_2\ \mathcal{R}_2\ T_3 \text{ for some } T_2\,\}$$
$$\text{and } Id^{+}(\mathcal{R}) = \begin{cases} \mathcal{R} & \mathcal{R} \neq \emptyset \\ \text{undefined} & \text{otherwise.} \end{cases}$$

This definition can be used to calculate a direct recursive characterization of consistent transitivity (see Appendix A), but we can also prove that it is equivalent to a combination of simpler operations.[17]

Proposition 20. *Evidence composition can be calculated as a series of interior operations, combining the information of both gradual types "in the middle" into a single type, formally :*

$$\langle S_1, S_{21}\rangle \fatsemi \langle S_{22}, S_3\rangle = \mathcal{I}[\![\pi_1(\mathcal{I}[\![S_1 \lesssim (S_{21} \sqcap S_{22})]\!]) \lesssim \pi_2(\mathcal{I}[\![(S_{21} \sqcap S_{22}) \lesssim S_3]\!])]\!]$$

where $S_{21} \sqcap S_{22} = \alpha(Id^{+}(\gamma(S_{21}) \cap \gamma(S_{22})))$.

A key observation here is that pairs of gradual types are but one arbitrary, albeit convenient, abstraction for subsets of the static subtyping relation. This analysis of evidence as abstractions of static subtyping fragments, and consistent transitivity as abstract relational composition, is critical to our analysis and improvement of evidence in Section 3.4. When developing evidence for AGTLC in Section 3.1, we also used pairs of types as the abstraction to be able to draw parallels between both designs, but because type equality is a simpler relation, one could use just a single gradual type instead to represent type equality evidence.

3.4 precise evidence for GTFL$_{\lesssim}$: bounded records and rows with forward completeness

Previous sections identify the need for a more precise abstraction for run-time evidence. This section delivers that abstraction and describes a general-purpose characterization of precision, forward-completeness, that meets our goals. Forward-completeness can be used as a design guideline for future AGT developments to guarantee that run-time checking (through evidence composition) is sufficiently precise.

We devise a notion of evidence that admits a precise consistent transitivity operation and preserves the type invariants implied by the concrete fragments of static subtyping that each evidence object represents. This type abstraction has some features in common with record-typing systems from the literature (Rémy, 1989), in particular polymorphic rows, though used here for run-time type checking. As it turns out, our representation of evidence still takes the form of pairs of abstractions of sets of types, but now with finer structure.

We replace gradual rows in evidence with a more precise gradual type abstraction, which we call *bounded records and rows* (BRR). We prove that evidence objects based

on BRR are precise using a general concept from abstract interpretation called *forward completeness*.

3.4.1 *Representing Optional Fields*

To motivate and introduce the structure of BRR, consider once again the failing example from Section 3.2.5. The key observation in that example is that in the following evidence composition,

$$(\langle[\mathsf{x} : \mathsf{Int}, \mathsf{y} : \mathsf{Bool}], [\mathsf{x} : \mathsf{Int}]\rangle \fatsemi \langle[\mathsf{x} : \mathsf{Int}], [?]\rangle) = \langle[\mathsf{x} : \mathsf{Int}, \mathsf{y} : \mathsf{Bool}], [?]\rangle$$

the second resulting gradual type represents too many possible static types, in particular the spurious cases $[y : \mathsf{Bool}]$ and $[x : \mathsf{Int}, y : \mathsf{Bool}]$. A precise analysis of the two composed evidence objects admits only two possible supertypes: $[\,]$ or $[x : \mathsf{Int}]$.

To faithfully capture this circumstance, BRR first introduces a distinction between *required* fields and *optional* fields. For instance, the BRR type $[\mathsf{x} : \mathsf{Int}_R]$ uses the R annotation to indicate that the x field is required: this BRR is equivalent to the $[\mathsf{x} : \mathsf{Int}]$ gradual record type (and static record type). In contrast, the BRR type $[\mathsf{x} : \mathsf{Int}_O]$ uses the O annotation to indicate that the x field is optional. Given the appropriate definition of concretization for BRR, we have $[\,] \in \gamma([\mathsf{x} : \mathsf{Int}_O])$ and $[\mathsf{x} : \mathsf{Int}\,] \in \gamma([\mathsf{x} : \mathsf{Int}_O])$, and no other static type. This BRR type precisely captures the intended record types in the example above, and as we might expect, the above evidence and composition can be replaced by the precise BRR-based version:

$$\begin{aligned}(\langle[\mathsf{x} : \mathsf{Int}_R, \mathsf{y} : \mathsf{Bool}_R], [\mathsf{x} : \mathsf{Int}_R]\rangle \fatsemi \langle[\mathsf{x} : \mathsf{Int}_R], [\mathsf{x} : \mathsf{Int}_O]\rangle)\\ = \langle[\mathsf{x} : \mathsf{Int}_R, \mathsf{y} : \mathsf{Bool}_R], [\mathsf{x} : \mathsf{Int}_O]\rangle\end{aligned}$$

Because the BRR abstraction is strictly more precise than the gradual row abstraction, the criteria set forth in Proposition 15 lead to more well-formed evidence objects. As a result, $\langle[\mathsf{x} : \mathsf{Int}_R], [\mathsf{x} : \mathsf{Int}_O]\rangle$ is well formed and, unlike gradual rows, it precisely captures the static subtype instances of interest.

3.4.2 *Representing Absent Fields*

Unfortunately, adding optional fields is not enough to achieve precision. Consider the following trio of BRR-based evidence:

$$\langle[\mathsf{x} : \mathsf{Int}_O, ?], [\mathsf{x} : \mathsf{Int}_O, ?]\rangle \fatsemi \langle[\mathsf{x} : \mathsf{Bool}_O, ?], [\mathsf{x} : \mathsf{Bool}_O, ?]\rangle \fatsemi \langle[\mathsf{x} : \mathsf{Int}_R], [\,]\rangle$$

BRR must somehow represent rows to subsume the expressiveness of gradual row types. However, evaluating this composition reveals the need for more precise information. Composing left-to-right yields $\langle[\mathsf{x} : \mathsf{Int}_O, ?], [\mathsf{x} : \mathsf{Int}_O, ?]\rangle$ because composing the first two yields $\langle[\mathsf{x} : \mathsf{Int}_O, ?], [?]\rangle$ which composes with the third to yield the above. On the other hand, composing them right-to-left fails! The reason is that the requirements of the second and third evidence objects are incompatible: the second can only have a Bool-typed x field, while the third demands an Int-typed one.

Closer inspection of the first two evidence objects reveals that after composing them, the resulting evidence object should remember that the super type position may *never have an* x *field*: the only instances of static subtyping that could have transitively composed were those lacking an x field. Then the fact that x must never be there is incompatible with requiring x to have Int type as in the third evidence.

This observation leads us to represent fields that are forbidden from appearing, which act as a hint for later compositions that should fail. We call these *necessarily absent* fields. Then the result of composing the first two evidence objects above is$\langle [\mathsf{x} : \mathsf{Int}_O, ?], [\mathsf{x} : \varnothing, ?] \rangle$. The BRR $[\mathsf{x} : \varnothing, ?]$ represents the set of record types with any fields *except* x, which can never appear. Composing this with $\langle [\mathsf{x} : \mathsf{Int}_R], [\,] \rangle$ should always fail. Adding both absent and optional fields suffices to ensure that evidence objects never lose information about plausible subtype relations, and that inconsistent constraints over fields never get reabsorbed into the unknown row ?.

3.4.2.1 *Differences with Wand (1991)*

While extending Rémy (1989), Wand (1991) discusses the idea of distinguishing whether a label in a record has a type or *is absent*, characterizing record types as total functions from labels to types or "absent". While superficially this notion seems connected to our work, they are semantically different: Wand's absent markers are not permanent, and thus a field marked as absent for a value may reappear later. It is exactly there where their usefulness lays: absent labels in polymorphic rows are a useful artifact to model functions that can extend records with new fields, as well as typing, for example, a record concatenation operator that can safely merge records whose label domains are disjoint. Our *necessarily absent* fields act as latent failure markers. A field marked as necessarily absent becomes permanently unobservable, as it has incompatible constraints: If the evidence object ε marks a field x as necessarily absent, any attempt to observe the contents via the projection $\varepsilon\mathbf{v}.\mathsf{x}$ for any record value $\mathbf{v}$ must result in a run-time type error.

3.4.3 *Bounded Rows and Bounded Records, Formally*

We now give a full formal definition of Bounded Records and Rows (BRR)[18], as well BRR-based evidence objects. The syntax of BRR follows:

$$M \in \textsc{Mapping}$$

$$\begin{array}{rcll} M & ::= & \varnothing \mid S_R \mid S_O & \text{(mappings for records and rows)} \\ S & ::= & \mathsf{Bool} \mid \mathsf{Int} \mid S \to S \mid ? & \text{(gradual types)} \\ & & \mid \left[\sum_{i=1}^{n} \ell_i : M_i\right] \mid \left[\sum_{i=1}^{n} \ell_i : M_i, ?\right] & \end{array}$$

As its name implies, the only interesting aspects of the BRR abstraction are its record and row type definitions. Each field maps to a type (qualified as required R or optional O), or to the absent field indicator ($\varnothing$). Individual static record types are represented in BRR by

simply marking all fields as required. BRR is a run-time abstraction, so its annotations do not affect GTFL$_{\lesssim}$ syntax.

Though technically we need missing field annotations only for rows and not for records, we opt for a uniform representation. A mapping $x : \varnothing$ in a record is redundant; similarly, a mapping $x : ?_O$ is redundant in a row. This redundancy has algorithmic advantages: we can extract a default mapping for labels in either a row or a record, which simplifies inductive rules in those cases where the domains of two types differ.

The notation $\sum_{i=1}^{n} \ell_i : M_i$ is equivalent to the previous notation $\overline{\ell_i : M_i}$, but avoids confusion when trying to distinguish shared subdomains in two record-like types. Empty record and row types are still allowed (i.e. n can be 0).

A concretization function (Figure 3.11) determines the meaning of BRR types. An intermediate notion of *decompositions* simplifies our definitions. Decompositions encode uniformity restrictions on sets of static types. These restrictions are made explicit by use of the generator function $\mathscr{C}$.

The interesting equations are those for records, which recursively describe sets of records that meet the constraints described informally above. Note that the concretization of non-empty bounded records is described in terms of the concretization of smaller bounded records.

Figure 3.12 defines the corresponding abstraction for bounded records and rows. The equations distinguish between relevant sets of static types. This abstraction function has a subtlety regarding its domain (which we do not explicitly name). The domain cannot be arbitrary sets of types. To see why, consider the set $\{ [\ell : \mathsf{Int}] \mid \ell \in \textsc{Label} \}$: What is the most precise abstraction of this set when $\textsc{Label}$ is an infinite set? The answer is that there is none, because this set features an infinite number of "optional" fields, and since bounded records have only finite fields, there is no best representative. The solution is to restrict the domain to hereditarily admit only collections of records that (1) have non-? bound for a finite set of field types; and (2) have either a finite set of potentially present fields (abstracts to a bounded record) or a finite set of absent fields (abstracts to a bounded row). The image of concretization satisfies these constraints, as do our operations on evidence. These finitary restrictions are analogous to the restrictions on the open sets of an infinite product topology (Munkres, 2000), and do not impose a burden upon programmers who write programs that are finite.

Theorem 21. *α and γ for bounded records and rows form a Galois Connection.*

Proof. Consequence of Soundness and Optimality Lemmas (see Appendix A.1). □

3.4.3.1 *Well-formed Evidence*

Using bounded records and rows, we develop a refined notion of evidence for GTFL$_{\lesssim}$. Figure 3.13 defines the inductive structure of well-formed BRR evidence. Its structure is somewhat analogous to the original, but with richer distinctions.

Decompositions

$$C \in \mathcal{P}(\textsc{Type}), C^{\varnothing} \in \mathcal{P}(\textsc{Type} \cup \{\varnothing\}), d \in \textsc{Decomp}$$

$$d \quad ::= \quad [\![\mathsf{Bool}]\!] \;\Big|\; [\![\mathsf{Int}]\!] \;\Big|\; [\![C \to C]\!] \;\Big|\; [\![?]\!] \;\Big|\; \left[\!\!\left[\sum_{i=1}^{n} \ell_i : C^{\varnothing}{}_i\right]\!\!\right] \;\Big|\; \left[\!\!\left[? \sum_{i=1}^{n} \ell_i : C^{\varnothing}{}_i\right]\!\!\right]$$

$\boxed{\mathscr{C} : (\textsc{Decomp}) \to \mathcal{P}(\textsc{Type})}$

$$\begin{aligned}
\mathscr{C}\,[\![\mathsf{Bool}]\!] &= \{\,\mathsf{Bool}\,\} \\
\mathscr{C}\,[\![\mathsf{Int}]\!] &= \{\,\mathsf{Int}\,\} \\
\mathscr{C}\,[\![C_1 \to C_2]\!] &= \{\; T_1 \to T_2 \mid T_1 \in C_1 \text{ and } T_2 \in C_2 \;\} \\
\mathscr{C}\left[\!\!\left[\sum_{i=1}^{n} \ell_i : C^{\varnothing}{}_i\right]\!\!\right] &= \left\{\; [\overline{\ell_i : T_i}] \;\middle|\; T_i \in C^{\varnothing}{}_i \;\right\} \text{ if } \varnothing \notin C_i^{\varnothing} \text{ for every } i \\
\mathscr{C}\left[\!\!\left[\ell_n : C^{\varnothing}{}_n \sum_{i=1}^{n-1} \ell_i : C^{\varnothing}{}_i\right]\!\!\right] &= \mathscr{C}\left[\!\!\left[\sum_{i=1}^{n-1} \ell_i : C^{\varnothing}{}_i\right]\!\!\right] \\
&\quad \cup \mathscr{C}\left[\!\!\left[\ell_n : (C^{\varnothing}{}_n \setminus \{\varnothing\}) \sum_{i=1}^{n-1} \ell_i : C^{\varnothing}{}_i\right]\!\!\right] \text{ if } \varnothing \in C^{\varnothing}{}_n \\
\mathscr{C}\left[\!\!\left[? \sum_{i=1}^{n} \ell_i : C^{\varnothing}{}_i\right]\!\!\right] &= \left\{ [\overline{\ell_i : T_i}\;\overline{\ell_j : T_j}] \;\middle|\; \begin{array}{l} [\overline{\ell_i : T_i}] \in \mathscr{C}\left[\!\!\left[\sum_{i=1}^{n} \ell_i : C^{\varnothing}{}_i\right]\!\!\right] \\ \text{and } [\overline{\ell_j : T_j}] \in \\ \qquad \mathscr{C}\left[\!\!\left[\sum_{j=1}^{m} \ell_j : \mathcal{P}\,(\textsc{Type} \cup \{\varnothing\})\right]\!\!\right] \\ \text{where } \overline{\ell_i} \textit{and} \overline{\ell_j} \text{ are disjoint.} \end{array} \right\}
\end{aligned}$$

$\boxed{\gamma : (\textsc{GType}) \to \mathcal{P}^{+}(\textsc{Type})}$

$$\begin{aligned}
\gamma\,(?) &= \textsc{Type} \\
\gamma\,(\mathsf{Bool}) &= \mathscr{C}\,[\![\mathsf{Bool}]\!] \\
\gamma\,(\mathsf{Int}) &= \mathscr{C}\,[\![\mathsf{Int}]\!] \\
\gamma\,(S_1 \to S_2) &= \mathscr{C}\,[\![\gamma(S_1) \to \gamma(S_2)]\!] \\
\gamma\left(\left[\sum_{i=1}^{n} \ell_i : M_i\, *\right]\right) &= \mathscr{C}\left[\!\!\left[* \sum_{i=1}^{n} \ell_i : \gamma^{M}\,(M_i)\right]\!\!\right]
\end{aligned}$$

$\boxed{\gamma^{M} : (\textsc{Mapping}) \to \mathcal{P}^{+}(\textsc{Type} \cup \{\varnothing\})}$

$$\begin{aligned}
\gamma^{M}\,(\varnothing) &= \{\varnothing\} \\
\gamma^{M}\,(S_R) &= \gamma(S) \\
\gamma^{M}\,(S_O) &= \gamma^{M}\,(\varnothing) \cup \gamma\,(S_R)
\end{aligned}$$

Figure 3.11: Decompositions and BRR Concretization Function

$$
\begin{array}{rcl}
\alpha\left(\mathscr{C}\left[\!\left[\mathsf{Bool}\right]\!\right]\right) & = & \mathsf{Bool} \\
\alpha\left(\mathscr{C}\left[\!\left[\mathsf{Int}\right]\!\right]\right) & = & \mathsf{Int} \\
\alpha\left(\mathscr{C}\left[\!\left[C_1 \to C_2\right]\!\right]\right) & = & \alpha\left(C_1\right) \to \alpha\left(C_2\right) \\
\alpha\left(\mathscr{C}\left[\!\left[* \sum_{i=1}^{n} \ell_i : {C^{\varnothing}}_i\right]\!\right]\right) & = & \left[\sum_{i=1}^{n} \ell_i : \alpha^M\left({C^{\varnothing}}_i\right) *\right] \\
\alpha\left(\varnothing\right) & \text{undefined} & \\
\alpha\left(C\right) & = & ?\ \text{otherwise}
\end{array}
$$

$$
\begin{array}{rcl}
\alpha^M\left(\{\varnothing\}\right) & = & \varnothing \\
\alpha^M\left(\{\varnothing\} \cup C\right) & = & \left(\alpha\left(C\right)\right)_O \text{ if } C \text{ is not empty} \\
\alpha^M\left(C\right) & = & \left(\alpha\left(C\right)\right)_R \text{ if } \varnothing \notin C
\end{array}
$$

Figure 3.12: BRR Abstraction Function

$\boxed{\vdash \varepsilon\ \mathbf{wf}}$ **Well-formed Evidence**

$$
\frac{S \in \{\mathsf{Int}, \mathsf{Bool}, ?\}}{\vdash \langle S, S\rangle\ \mathbf{wf}}
\qquad
\frac{\vdash \langle S_{21}, S_{11}\rangle\ \mathbf{wf} \quad \vdash \langle S_{12}, S_{22}\rangle\ \mathbf{wf}}{\vdash \langle S_{11} \to S_{12}, S_{21} \to S_{22}\rangle\ \mathbf{wf}}
\qquad
\begin{array}{l} D(\cdot) = \varnothing \\ D(?) = ?_O \end{array}
$$

$$
\frac{\begin{array}{c}
\langle *_1, *_2\rangle \neq \langle \cdot, ?\rangle \qquad \forall i, \vdash \langle M_{i1}, M_{i2}\rangle\ \mathbf{wf} \\
\forall j, \vdash \langle M_j, D(*_2)\rangle\ \mathbf{wf} \qquad \forall k, \vdash \langle D(*_1), M_k\rangle\ \mathbf{wf} \\
\text{where } \overline{\ell_i}, \overline{\ell_j}, \text{ and } \overline{\ell_k} \text{are disjoint.}
\end{array}}{\vdash \left\langle \left[\sum_{i=1}^{n} \ell_i : M_{i1} \sum_{j=1}^{m} \ell_j : M_j *_1\right], \left[\sum_{i=1}^{n} \ell_i : M_{i2} \sum_{k=1}^{o} \ell_k : M_k *_2\right]\right\rangle\ \mathbf{wf}}
$$

$\boxed{\vdash \langle M, M\rangle\ \mathbf{wf}}$ **Well-formed Mappings**

$$
\frac{}{\vdash \langle M, \varnothing\rangle\ \mathbf{wf}}
\qquad
\frac{\vdash \langle S_1, S_2\rangle\ \mathbf{wf}}{\vdash \langle {(S_1)}_R, {(S_2)}_R\rangle\ \mathbf{wf}}
\qquad
\frac{S_1 \sqsubseteq S_3 \quad \vdash \langle S_1, S_2\rangle\ \mathbf{wf}}{\vdash \langle {(S_3)}_*, {(S_2)}_O\rangle\ \mathbf{wf}}
$$

Figure 3.13: BRR$_{\lesssim}$'s definition of well-formed evidence

3.4.4 *Absent Labels Enable Sound Optimizations*

We motivate the introduction of bounded records and rows to guarantee that programs with inconsistent ascriptions always fail. More broadly, precision could support improvements to program optimizations such as inlining and pre-composing evidence objects. Consider the following program:

$$(\lambda x : [?].\,(\mathsf{if\ true\ then}\ (\mathsf{if\ true\ then}\ x\ \mathsf{else}\ x :: [l : \mathsf{Int}, ?])\ \mathsf{else}\ x :: [l : \mathsf{Bool}, ?]))$$

This program relies on if branching to generate uses of consistent subtype join, which generate evidence objects that use the optional annotation. This program produces an error when given as an argument any record that has an l mapping, because the ascriptions impose inconsistent constraints: the function body must have type $[l : \mathsf{Bool}_O, ?]$ in the outermost if , and $[l : \mathsf{Int}_O\, ?]$ in the innermost if , thus x must go through the composition of evidence on both types, reaching an inconsistency whenever the label is present.

While the sole introduction of optional fields would suffice to run this program properly, consider now an optimizing compiler that performs constant propagation in this function. The body of the function might then be optimized to $\lambda x.\varepsilon_2\varepsilon_1 x$, with $\varepsilon_2 = \langle [l : \mathsf{Bool}_O, ?], [l : \mathsf{Bool}_O, ?] \rangle$ and $\varepsilon_1 = \langle [l : \mathsf{Int}_O, ?], [l : \mathsf{Int}_O, ?] \rangle$. A more advanced optimizing compiler could try to perform this evidence composition ahead of time. Unfortunately, unless we introduce absent labels as in BRR, the only possible composition would be to coalesce the label l into the row portion to generate the evidence pair $\langle [|?], [|?] \rangle$. This "optimization" changes the behaviour of our program, as arguments with a mapping for l are now accepted instead of producing an error. To achieve full precision and soundness in the presence of optimizations like the above, we need absent labels.

3.5 FORWARD COMPLETENESS AS A KEY TO PRECISION

The previous section delves into the shortcomings of GTFL$_{\lesssim}$'s evidence abstraction, diagnoses some evident information loss, and devises a new abstraction that retains the relevant information. Are these improvements sufficient? An example-driven approach can drive us closer to a solution, but ultimately, we need more rigorous and comprehensive confirmation, which we now provide. Moreover, we do so by generalizing beyond GTFL$_{\lesssim}$, seeking sufficient criteria that can apply to future applications of AGT, regardless of the particulars of the type discipline or gradualization.

How does one confirm that the evidence abstraction is precise enough to protect type-based invariants? In the context of abstract model-checking, Giacobazzi and Quintarelli (2001) introduce the concept of *forward completeness*,[19] which is dual to the concept of *(backward) completeness* that arises more naturally for abstract interpretation-based static analysis. The idea applies to any abstract function, but we present it here in domain-specific terms.

Definition 12 (Forward completeness)**.** *Let ε denote evidence objects, $\fatsemi$ denote consistent transitivity, and ; denote relational composition over fragments of static subtyping. Then $\fatsemi$ is*

forward complete *with respect to its evidence abstraction if the operation does not introduce any noise: The set of all subtyping tuples in the concretization of the result is exactly the same as if we operated directly by relational composition on the concretizations of the original evidences. Formally,* $\gamma^{<:}(\varepsilon_1) \,;\, \gamma^{<:}(\varepsilon_2) = \gamma^{<:}(\varepsilon_1 \fatsemi \varepsilon_2)$ *for any two evidence objects* ε_1 *and* ε_2*.*

Soundness of $\fatsemi$ with respect to relational composition ; implies $\gamma^{<:}(\varepsilon_1) \,;\, \gamma^{<:}(\varepsilon_2) \subseteq \gamma^{<:}(\varepsilon_1 \fatsemi \varepsilon_2)$, which ensures that $\fatsemi$, operating on abstract evidence objects, sufficiently overapproximates the behaviour of ; on the meanings of those objects. Forward completeness implies that the reverse-containment also holds, which means that $\fatsemi$ *exactly* approximates ; for abstract objects. This means that $\fatsemi$ is a perfect stand-in for ; *if we need only consider sets in the image of evidence concretization*. In the case of AGT, such is exactly the case: evidence is initially introduced in terms of consistent subtyping relations among gradual types, which are even less precise than evidence objects.

Lemma 2 (Relational Composition is closed wrt concretization)**.** *The result of relational composition among the concretization of well-formed evidence objects is either an empty set or it produces a set that is the image of the concretization of some well-formed evidence. Formally, for any* $\vdash \varepsilon_1$ **wf** *and* $\vdash \varepsilon_2$ **wf**, *either* $\gamma^{<:}(\varepsilon_1) \,;\, \gamma^{<:}(\varepsilon_2) = \emptyset$, *or there exists an* $\vdash \varepsilon_3$ **wf** *such that* $\gamma^{<:}(\varepsilon_1) \,;\, \gamma^{<:}(\varepsilon_2) = \gamma^{<:}(\varepsilon_3)$.

Proof. By double induction over $\vdash \varepsilon$ **wf** and $\vdash \langle M, M \rangle$ **wf** for ε_1 and ε_2. □

Theorem 22. *In BRR,* $\fatsemi$ *is forward complete.*

Proof. Consequence of Lemma 2. □

Forward completeness ensures that reasoning about transitivity of subtyping is as precise as operating directly with fragments of subtyping, as alluded to in Section 3.3. This precision matters most when considering precise enforcement of type invariants.

3.6 NOTES

1. For clarity, these examples use a let syntax with type annotations. Our formal semantics exhibit the same concepts, albeit using a necessarily less transparent representation.

2. I follow the naming convention of Ingerman (1967), acknowledging that the Sanskrit grammar of Pānini uses a notation equivalent to the syntax notation introduced into programming languages by Backus while anteceding the latter by more than two millennia.

3. Well-modedness originates from the logical programming literature, and a mode analysis ensures that when assuming that the inputs are "ground" (that is, resolved), we can prove that the outputs are also "ground".

4. The particular case of type equality is unfortunate, because every type is only related to one other type. Because there is only one type to relate against, one could think as possible to assign to the equality predicate an input mode on the left, and an output mode on the right and recover well-modedness for the static language. This approach does not scale to the gradual version of the language. We intend to preserve the same modes between the static and the gradual versions of the language, and in the gradual language, there is more than one gradual type to relate against, so the predicate has to be checked and not computed. All types need to be considered as inputs to the predicate check once in the gradual domain, so the argument remains.

5. I say "less imprecise" to read the arguments in the order they are written left-to-right, even though the wording produces a double negative. When reading $S_1 \sqsubseteq S_2$ one must remember that S_1, which is on the left, can be "more precise" than the S_2 that is on the right. This goes against the usual reading of a partial order, where for example, $1 < 3$ is read as "1 is less than 3". For $S_1 \sqsubseteq S_2$, we could either say that "S_1 is less imprecise than S_2", that "S_1 is more precise than S_2", or that "S_2 is less precise than S_1". I choose the first option.

6. In general, when two functions form a Galois connection, they are always sound and optimal with respect to each other. For the opposite direction, it suffices that the functions be monotonic and injective.
   Injectivity can be weakened: it also suffices if whenever two function outputs are related by the partial order of their codomain, then the function inputs must also be related by the partial order of their domain. This means that we can prove that the concretization and abstraction functions form a Galois Connection from the fact that soundness and optimality holds, even for gradual type designs like type-and-effect systems (Bañados Schwerter, Garcia, and Tanter, 2014, 2016), which are not injective.

7. The guarantee of uniqueness and existence depends on the existence of meets on the concrete domain, as we discuss in the following paragraph, and on the concretization function preserving meets.

8. Unlike Garcia, Clark, and Tanter (2016), we define α to be a *total* function over a family of (non-empty) sets of types, rather than a *partial* function over *arbitrary* sets of types $\mathcal{P}(\textsc{Type})$. This approach pushes partiality into the collecting operators (*cf.* cod^+ below), rather than the Galois connection itself. We find this approach more general, intuitive, and mathematically pleasant.

9. We discuss how to generate this derivation in Section 3.1.8.

10. AGTLC ARL's safety follows from how its dynamic semantics were calculated, using AGT, from the safety proof of STLC.

11. The inductive definition of initial evidence for AGTLC is a subset of the rules we present for the inductive definition of initial evidence for GTFL$_{\lesssim}$ in Figure A.6

12. Throughout we use overlines to denote zero or more repetitions, $+$-annotated overlines to denote one or more repetitions, and $[\overline{l : S}, *]$ to simultaneously denote both traditional record types $[\overline{l : S}]$ and gradual row types $[\overline{l : S}, ?]$.

13. We also define composition directly in Appendix A

14. This is most evident in the fact that consistent subtyping is not transitive.

15. An inductive definition of initial evidence for is presented in Figure A.6.

16. The typing and elaboration rules for let can be deduced from the rules for lambda and application.

17. Garcia, Clark, and Tanter (2016) erroneously omits the outermost instance of $\mathcal{I}$ in their definition.

18. While not *Rows* in the sense of Wand (1991), *Bounded Rows* are derived from Gradual Rows introduced by (Garcia, Clark, and Tanter, 2016). We keep the *row* designation to highlight the connection with the latter paper.

19. Sometimes called gamma-completeness or exactness

Life must be understood backward. But [...] it must be lived forward.

— Kierkegaard (1843)

4 THE ESSENCE OF GALOIS SLICING

Sometimes one writes programs that are just bad. The language semantics specifies some output, but that output was not what the programmer wanted. That result, though obtained by following the rules of the semantics, is unexpected for the programmer. But the language is just doing what it said it would do: it is the program that is wrong and needs debugging.

One first step for debugging is to obtain and use an evaluation trace. An evaluation trace justifies the computations of the language semantics, detailing the relevant steps to reach the output result. But executing the program may involve many complicated computations, making the trace too long or too detailed to understand. How does a programmer use a trace when debugging? Programmers use a trace to triangulate a possible cause of a failure, focusing on specific portions of a program and checking whether a computation is as expected or not. These focus points are associated with some intermediate evaluation states in the trace that the programmer can use, for example, to observe whether the argument to a function call is "wrong": the trace might say at some point that the program calls $f\ (10)$ while the programmer knows it should be $f\ (-10)$ instead. The programmer found an inconsistency between the trace and the behaviour they want the program to have.

From finding an inconsistency in the trace, a programmer may ask questions: "Does this (undesired) argument affect the final result?", if it does, they may also ask "Where in the original program does this argument come from?". Since that argument may be the result of a complicated computation, the more precise question a programmer might ask is "What part of the original program was involved in computing this argument?".

As discussed in Section 2.2.1, these are questions that dynamic program slicing (Korel and Laski, 1988; Weiser, 1981) can answer. A *slice* is a portion of an execution state in the program trace. Programmers can use slices to represent the program parts they are interested in. A slicing system provides two functions to connect a slice of a particular intermediate state with other states: Forward slicing and backward slicing. *Forward slicing* computes the parts of some later state impacted by a slice, while *backward slicing* computes the parts of a previous state that impact a slice. When a language includes a slicing tool, programmers can take a trace and ask these questions at any point of the trace and about any particular portion of interest.

It is no small feat to design slicing tools for an arbitrary programming language. Much like AGT used Galois connections to systematically design gradual languages from a statically typed language, *Galois slicing* (Perera, Acar, et al., 2012; Perera, Garg, and Cheney, 2016; Ricciotti et al., 2017) uses Galois connections to systematically design dynamic slicing procedures from a runtime semantics.

This chapter describes the essence of Galois slicing, synthesizing ideas introduced in the literature into a tutorial conveying the essence of the technique. My goal is for the reader to become not only familiar with the details of the Galois slicing framework, but also for them to learn how to apply its recipe to their own language designs so as to provide slicing-based debugging tools for their language users. To this end, I show how to use the Galois slicing recipe by example in the context of structural operational semantics and reduction semantics, justifying decisions at every step in a calculational way. I address increasingly complicated languages:

- I begin with a tutorial introducing the basic concepts of Galois slicing in a relatively simple language, a structural operational semantics for a calculator that supports random number generation. This (almost trivial) language lets us focus on conveying the fundamental concepts underlying Galois slicing, and clarifying their roles in systematically designing the parts of a slicing system.

- I extend the language to include recursive functions, which allows us to write more interesting programs. Previous work uses environments and closures when representing identifier bindings and functions as values. But most structural operational semantics and reduction semantics rely on an auxiliary *substitution* function to model identifiers and their bindings. I take this opportunity to introduce Galois slicing for substitution, which was not previously described and serves to discuss slicing of auxiliary functions.

- I extend the language to include mutable state. Mutable state introduces new problems for slicing. Slices now include both the program code and the memory store. The contents of a store location can be marked as relevant separately, even when the whole program is marked as irrelevant in the slice. When the program is marked as irrelevant, backward slicing must preserve the program operations generating the store contents marked as relevant. Outside those operations, backward slicing only has to preserve "minimal program structure" to access the operations of interest that update the store.

- I extend the language with first class continuations. My goal here is twofold: first, continuations are included to show that the framework scales up to accommodate arbitrary control flow and advanced language features. Second, I use this language to motivate introducing slicing for reduction semantics, which forms the basis for most AGT developments, including those discussed in Chapter 3.

I present most of these language semantics using structural operational semantics (Plotkin, 1981), but this systematic approach also works for reduction semantics. Previous approaches have focused on natural semantics (Perera, Acar, et al., 2012) and transition systems for the π-calculus (Perera, Garg, and Cheney, 2016). I discuss structural operational semantics and reduction semantics in this chapter and connect one of these languages to natural semantics in Section 4.4.6. I direct readers interested in the formal details of the connection between slicing natural semantics and structural operational semantics to Appendix C.

I do not include type systems for any language discussed in this chapter. A type system serves to distinguish legal programs and allow those to run. Type safety ensures the

Syntax

$$
\begin{array}{rcll}
t \in \textsc{Term} & ::= & v \mid t + t \mid \mathsf{rand} & \text{(terms)} \\
F \in \textsc{Frame} & ::= & \square + t \mid v + \square & \text{(frames)}
\end{array}
\qquad
\begin{array}{rcll}
v \in \textsc{Value} & ::= & n & \text{(values)} \\
r \in \textsc{Redex} & ::= & v + v \mid \mathsf{rand} & \text{(redexes)}
\end{array}
$$

$\boxed{r \rightsquigarrow t}$ **Notions of Reduction**

$$
\begin{array}{rcll}
n_1 + n_2 & \rightsquigarrow & n_3 & \text{where } n_3 = n_1 + n_2 \\
\mathsf{rand} & \rightsquigarrow & n & (n \text{ is any number})
\end{array}
$$

$\boxed{t \mapsto t}$ **Contextual Reduction**

$$
\frac{r \rightsquigarrow t}{r \longmapsto t}
\qquad
\frac{t_1 \longmapsto t_2}{F[t_1] \longmapsto F[t_2]}
$$

Figure 4.1: The CARL language

runtime semantics is complete for all legal programs. But (dynamic) slicing assumes the previous existence of an evaluation trace: the program was already run. Because slicing depends on a program already run, type checking and slicing are orthogonal problems. I avoid types in this chapter to focus the reader's attention on slicing.

4.1 EXAMPLE 1: A CALCULATOR (WITH RANDOMNESS)

I first introduce the language where I apply the Galois slicing recipe to serve as a concrete example while discussing the recipe in the following section.

I present the CARL language in Figure 4.1.[1] This language is by no means complicated but suffices to provide a complete development for a slicing system using the recipe, without bogging down in secondary details that arise in more interesting languages.

CARL has primitive operations on numbers (represented by addition) and a random number generator "rand". rand provides a concrete reason to need the trace. If the calculator did not have rand, one may be tempted to define slicing without appealing to the trace. But skipping the trace would be deceiving: the trace could only be "skipped" because execution is deterministic, and the only possible trace can be reconstructed whenever needed. rand makes the need for a specific trace unavoidable.

Evaluation in CARL follows a left-to-right evaluation strategy for subexpressions, like in all the language examples I discuss. Its structural operational semantics (the "Contextual Reduction" relation) has one rule for immediate reduction and a structural rule that allows reduction inside the hole of a frame. This standard trick lets us have a single rule for structural reduction, isolating the complexity in the definition of frames.

4.2 A HIGH-LEVEL VIEW OF A SLICING SYSTEM

Before diving into the details of slicing for CARL, let us discuss slicing at a high level. A slicing system combines a representation for slices with algorithms to answer forward and backward slicing queries. The goal of Galois slicing is to design a program slicing system for arbitrary program traces T. A trace represents the evaluation of a program for some arbitrary number of steps. I call each of these steps a *program execution state*. A program trace T is a derivation of the transitive reflexive closure of the contextual reduction relation ($\longmapsto^*$).

To use a slicing system, programmers need a language to formulate their questions. Programmers require an interface to:

1. Select one particular execution state of interest. This selection sets the context for the programmers' slicing questions. Once the programmer picks a particular state t, the original trace T can be split into two sub traces for asking questions, either about the future or the past with respect to the state of interest: one trace represents the m steps of computation in the trace *before* t and another represents the n steps of computation *after* t.

2. Select a portion of interest of the program state at hand, distinguishing (or *slicing*) those parts of the program state from other parts considered irrelevant. This allows the programmer to ask slicing questions over a section of interest of an execution state. Following the literature, each portion is called a *slice*. For every state t, there exists a set of all its possible slices, defined by the slicing system design. To account for cases where the complete state t is relevant, every complete program state also counts as a slice with nothing sliced away. Whenever terms represent complete program states, it is the case that $\textsc{Term} \subseteq \textsc{Slice}$.

What does a programmer gain from access to a slicing system? A programmer can use the slicing system obtained with Galois slicing to:

3. Answer their forward slicing questions. To answer these questions, the system must be able to replay a slice of the state t at step m for any number of steps n' of the n steps left after t in the trace. Replaying computes a slice of the state t' at step $m + n'$, with the aim of answering "What portion/slice $\mathring{t}'$ of state t' is affected by portion $\mathring{t}$ of state t?". This replay function is called *forward slicing* (fwd).

4. Answer their backward slicing questions. To answer these questions, the system must be able to rewind a slice of the state t at step m for any number of steps m' of those m steps before t in the trace. Rewinding acts over the trace in the opposite direction to replay, computing a new slice of the state t' at step $m - m'$, answering the question "What portion/slice $\mathring{t}'$ of state t' affects portion $\mathring{t}$ of state t?". This rewind function is called *backward slicing* (bwd).

I present the Galois slicing framework as an approach to develop slicing systems that can answer programmers' forward and backward slicing questions. As an example, Figure 4.2 summarizes a slicing system for CARL, the language introduced in Figure 4.1. Before discussing how to design a slicing system, let's first see what a slicing system looks like in practice and how it answers programmer questions.

4.2.1 *A slicing example*

What counts as a trace for CARL? Consider the program $(1 + \mathsf{rand}) + (2 + 5)$, here is a trace of its execution:[2]

$$(1 + \mathsf{rand}) + (2 + 5) \underset{\mathcal{S}1}{\longmapsto} (1 + 34) + (2 + 5) \underset{\mathcal{S}2}{\longmapsto} 35 + (2 + 5) \underset{\mathcal{S}3}{\longmapsto} 35 + 7 \underset{\mathcal{S}4}{\longmapsto} 42$$

This trace is short, so one may be able to figure out answers to evaluation questions directly by inspecting the trace, but that is definitely not always the case. Consider this example as a proxy for longer traces and bigger programs. Each individual trace step is given a unique name $\mathcal{S}n$ for later reference.

Galois slicing was originally formulated over big-step traces, so programmers could only take slices of either the original program or the final result. Here I present slicing in terms of small-step traces, defining multi-step slicing in terms of single-step slicing. When slicing is performed at small-step granularity, it can be used to identify relationships between multiple breakpoints at different steps of the evaluation trace. Suppose the programmer did not expect the result of this program to be 42. A developer can pick any execution step of the trace as a breakpoint, for example the state after $\mathcal{S}3$, $(35+7)$. The developer may then recognize the 7 as a correct input but the 35 on the left as suspicious.

Our example's developer then can ask "What part of the original program computes that 35?". This question can be formulated by slicing the state to the portion of interest, in this case $35+\blacksquare$ ($\blacksquare$ being a marker for an irrelevant subexpression, to be read as "irrelevant"),[3] and then answered by applying backward slicing up to the original program state. A slicing tool achieves this by repeatedly applying single-step backward slicing through the trace until it reaches the original program state:

$$\underset{\mathcal{S}1\mapsto^*\mathcal{S}3}{\mathsf{bwd}^*} = \underset{\mathcal{S}1}{\mathsf{bwd}}\left(\underset{\mathcal{S}2}{\mathsf{bwd}}\left(\underset{\mathcal{S}3}{\mathsf{bwd}}\,(35+\blacksquare)\right)\right) = (1+\mathsf{rand})+\blacksquare$$

The programmer can then see that a call to the random number generator (and an addition) reached that value of 35. Perhaps they did not expect a call to rand. As they explore this program, developers can also ask "What part of the final result does not depend on that rand call to the number generator?" by slicing the initial program to indicate as relevant the entire program except for the rand, *i.e.*, $(1+\blacksquare)+(2+5)$.

To answer this question, we need to obtain a slice of the final program state. The tool obtains such a slice by repeatedly applying a single-step forward slicing function until reaching the final state:

$$\underset{\mathcal{S}1\mapsto^*\mathcal{S}4}{\mathsf{fwd}^*}\,((1+\blacksquare)+(2+5)) = \blacksquare$$

Because this language is simple, the final state is touched by all parts of the original program, including rand. However, by looking at the step-by-step computation of forward slicing, one can observe that, after the third step, the 7 in the penultimate state does not depend on the randomness in the original program.

$$\begin{aligned}
\underset{\mathcal{S}1\mapsto^*\mathcal{S}3}{\mathsf{fwd}^*}\,((1+\blacksquare)+(2+5)) &= \underset{\mathcal{S}3}{\mathsf{fwd}}\left(\underset{\mathcal{S}2}{\mathsf{fwd}}\left(\underset{\mathcal{S}1}{\mathsf{fwd}}\,((1+\blacksquare)+(2+5))\right)\right)\\
&= \underset{\mathcal{S}3}{\mathsf{fwd}}\left(\underset{\mathcal{S}2}{\mathsf{fwd}}\,((1+\blacksquare)+(2+5))\right)\\
&= \underset{\mathcal{S}3}{\mathsf{fwd}}\,(\blacksquare+(2+5))\\
&= \blacksquare+7
\end{aligned}$$

The slicing tool does not choose sides and claim some part to be correct or incorrect. That choice is up to the programmer. The slicing system is as an exploratory debugging tool that developers can use to understand relationships between portions of the program trace.

4.2.2 *Defining program slices*

The first step in formally specifying a slicing system is to define a notion of slices of run-time program states. Slices specify the portions of programs that can be "sliced out", i.e., marked as irrelevant in a particular program. For this simple language, slices of terms suffice to represent all execution states.[4]

A common design is to accept any arbitrary subexpression as a candidate for slicing, so that the definition of slicing follows the same inductive structure as terms, but with the addition of a new possible base case: ■ (read as "irrelevant"). The ■ slice indicates a subexpression that is considered irrelevant. The complete definition of CARL term slices is presented in Figure 4.2.

This definition of term slices induces an information partial order among slices. One slice is *less informative* than another if it contains less information, but otherwise is structurally the same. That order is defined by the following set of rules:

$$\frac{}{\blacksquare \sqsubseteq \mathring{t}} \qquad \frac{}{n \sqsubseteq n} \qquad \frac{}{\mathsf{rand} \sqsubseteq \mathsf{rand}} \qquad \frac{\mathring{t}_1 \sqsubseteq \mathring{t}'_1 \quad \mathring{t}_2 \sqsubseteq \mathring{t}'_2}{\mathring{t}_1 + \mathring{t}_2 \sqsubseteq \mathring{t}'_1 + \mathring{t}'_2}$$

Lemma 3 ($\sqsubseteq$ is a partial order)**.** *The $\sqsubseteq$ relation is reflexive, transitive, and antisymmetric.*

Since terms count as slices, the same relation applies to terms:

Lemma 4 (Terms are slices)**.** *For every $t \in$ Term, $t \in$ Slice.*

The partial order specifies what counts as a valid slice of a term. In particular, for every term t a set $\textsc{slices}[t] \subset \textsc{Slice}$ is specified as follows:[5]

Definition 13 (Valid slices of a term)**.** *For every $t \in$ Term the set of valid slices of t, $\textsc{slices}[t]$, is defined as:*

$$\textsc{slices}[t] = \left\{ \mathring{t} \mid \mathring{t} \sqsubseteq t \right\}$$

4.2.3 *The slicing framework*

The language designer now defines two functions, forward and backward slicing. These functions work together to extract slices from an evaluation trace. The trace connects some input with some output. Forward slicing produces a slice of the output from a slice of the input, and backward slicing produces a slice of the input from a slice of the output. The next step is to specify the expected behaviour of both functions.

In general a slicing system can be defined for any logical judgment represented as derivation trees, although for now I focus on binary relations among execution state representations . For CARL, execution states are completely represented by terms, so I focus on $R \subseteq (\textsc{Term} \times \textsc{Term}))$. I consider derivation trees as a trace of the logical

Syntax

$$
\begin{array}{rcll}
\mathring{t} \in \textsc{Slice} & ::= & \mathring{v} \mid \mathring{t} + \mathring{t} \mid \mathsf{rand} \mid \blacksquare & \text{(term slices)} \\
\mathring{v} \in \textsc{VSlice} & ::= & n \mid \blacksquare & \text{(value slices)} \\
\mathring{F} \in \textsc{FSlice} & ::= & \square + \mathring{t} \mid \mathring{v} + \square & \text{(frame slices)}
\end{array}
$$

$\boxed{\underset{r \rightsquigarrow t}{\overset{\mathcal{D}}{\mathsf{fwd}}} : \textsc{slices}[r] \to \textsc{slices}[t]}$ **Notions of forward slicing**

$$
\frac{n_3 = n_1 + n_2}{\underset{n_1+n_2 \rightsquigarrow n_3}{\mathsf{fwd}}\ (n_1 + n_2) = n_3}
\qquad
\frac{\begin{array}{c} n_3 = n_1 + n_2 \\ \mathring{t}_1 = \blacksquare \text{ or } \mathring{t}_2 = \blacksquare \end{array}}{\underset{n_1+n_2 \rightsquigarrow n_3}{\mathsf{fwd}}\ (\mathring{t}_1 + \mathring{t}_2) = \blacksquare}
\qquad
\frac{n_3 = n_1 + n_2}{\underset{n_1+n_2 \rightsquigarrow n_3}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare}
$$

$$
\frac{n \text{ is any number}}{\underset{\mathsf{rand} \rightsquigarrow n}{\mathsf{fwd}}\ (\mathsf{rand}) = n}
\qquad
\frac{n \text{ is any number}}{\underset{\mathsf{rand} \rightsquigarrow n}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare}
$$

$\boxed{\underset{t \longmapsto t'}{\overset{\mathcal{D}}{\mathsf{fwd}}} : \textsc{slices}[t] \to \textsc{slices}[t']}$ **Contextual forward slicing**

$$
\frac{\underset{t_1 \rightsquigarrow t_2}{\mathsf{fwd}}\ (\mathring{t}_1) = \mathring{t}_2}{\underset{t_1 \longmapsto t_2}{\mathsf{fwd}}\ (\mathring{t}_1) = \mathring{t}_2}
\qquad
\frac{\underset{t_1 \longmapsto t_2}{\mathsf{fwd}}\ (\mathring{t}_1) = \mathring{t}_2 \quad \mathring{F} \sqsubseteq F}{\underset{F[t_1] \longmapsto F[t_2]}{\mathsf{fwd}}\ (\mathring{F}[\mathring{t}_1]) = \mathring{F}[\mathring{t}_2]}
\qquad
\frac{\underset{t_1 \longmapsto t_2}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare}{\underset{F[t_1] \longmapsto F[t_2]}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare}
$$

$\boxed{\underset{r \rightsquigarrow t}{\overset{\mathcal{D}}{\mathsf{bwd}}} : \textsc{slices}[t] \to \textsc{slices}[r]}$ **Notions of backward slicing**

$$
\frac{n_3 = n_1 + n_2}{\underset{n_1+n_2 \rightsquigarrow n_3}{\mathsf{bwd}}\ (n_3) = n_1 + n_2}
\qquad
\frac{n_3 = n_1 + n_2}{\underset{n_1+n_2 \rightsquigarrow n_3}{\mathsf{bwd}}\ (\blacksquare) = \blacksquare}
$$

$$
\frac{n \text{ is any number}}{\underset{\mathsf{rand} \rightsquigarrow n}{\mathsf{bwd}}\ (n) = \mathsf{rand}}
\qquad
\frac{n \text{ is any number}}{\underset{\mathsf{rand} \rightsquigarrow n}{\mathsf{bwd}}\ (\blacksquare) = \blacksquare}
$$

$\boxed{\underset{t \longmapsto t'}{\overset{\mathcal{D}}{\mathsf{bwd}}} : \textsc{slices}[t'] \to \textsc{slices}[t]}$ **Contextual backward slicing**

$$
\frac{\underset{t_1 \rightsquigarrow t_2}{\mathsf{bwd}}\ (\mathring{t}_2) = \mathring{t}_1}{\underset{t_1 \longmapsto t_2}{\mathsf{bwd}}\ (\mathring{t}_2) = \mathring{t}_1}
\qquad
\frac{\underset{t_1 \longmapsto t_2}{\mathsf{bwd}}\ (\mathring{t}_2) = \mathring{t}_1 \quad \mathring{F} \sqsubseteq F}{\underset{F[t_1] \longmapsto F[t_2]}{\mathsf{bwd}}\ (\mathring{F}[\mathring{t}_2]) = \mathring{F}[\mathring{t}_1]}
\qquad
\frac{\underset{t_1 \longmapsto t_2}{\mathsf{bwd}}\ (\blacksquare) = \blacksquare}{\underset{F[t_1] \longmapsto F[t_2]}{\mathsf{bwd}}\ (\blacksquare) = \blacksquare}
$$

Figure 4.2: Galois slicing definitions for the CARL language.

judgment. For any relation R, the designer defines a family of functions that operate on the relation traces, as each derivation of $a\ R\ b$ requires both forward slicing and backward slicing between the slices of a and the slices of b, with the following signatures:

$$\underset{aRb}{\mathsf{fwd}} : \textsc{slices}[a] \to \textsc{slices}[b] \qquad\qquad \underset{aRb}{\mathsf{bwd}} : \textsc{slices}[b] \to \textsc{slices}[a]$$

These functions are indexed by a particular derivation of a trace of $a\ R\ b$, although the notation here skips this detail for brevity. Each function is defined only for the valid slices of the input and output present in a particular trace, $\textsc{slices}[a]$ and $\textsc{slices}[b]$. Because these functions depend on a specific program trace, they are not defined in general for every possible syntactically valid slice in isolation ($\mathring{t} \in \textsc{Slice}$).

Definition 14 (Least requirements for forward slicing)**.** *Forward slicing for a trace of* $(a\ R\ b)$ *is a function from slices of* a *to slices of* b *that satisfies two constraints:*

1. ***Conservative extension****: if the input to forward slicing does not slice away anything, the forward slicing function must replicate the output from the trace. Formally,* $\underset{aRb}{\mathsf{fwd}}(a) = b$.

2. ***Monotonicity****: if one input to forward slicing has more information than another input slice, applying the forward slicing function must at least not lose information on the output slice. Formally, for any two slices of* a *such that* $\mathring{t}_1 \sqsubseteq \mathring{t}_2$, $\underset{aRb}{\mathsf{fwd}}(\mathring{t}_1) \sqsubseteq \underset{aRb}{\mathsf{fwd}}(\mathring{t}_2)$.

   The intuition behind monotonicity is that the addition of relevant information on the input should at least not slice away any relevant information on the output that was already presented.

These minimum specification requirements for forward slicing were introduced by Perera, Acar, et al. (2012).

The semantics of CARL relies on three relations, the notions of reduction relation ($\rightsquigarrow$), the contextual reduction relation ($\longmapsto$), and the transitive reflexive closure of the latter ($\longmapsto^*$). Each requires their own forward and backward slicing functions. Section 4.2.1 shows an example of derivation trees for these relations used as traces, even though the dependency on $\rightsquigarrow$ is left implicit for exposition purposes.

4.2.4 *Least requirements for backward slicing*

The results of forward and backward slicing must be compatible with each other. Programmers might expect these functions to mirror each other. Whenever the programmer uses forward slicing to narrow down the parts of a trace affected by some slice, they may expect that backward slicing that output slice would produce the very slice of the input that they started from. Similarly, if they backward slice first and then forward slice, they may expect to return to the same result they began from. This unfortunately is not always the case, as such mirroring functions may not exist.

Proposition 23 (Intuitive, but sometimes impossible specification for backward slicing)**.** *For any particular program trace, forward and backward slicing mirror each other. Formally, forward and backward slicing form a bijection, so that backward slicing is the inverse function*

to forward slicing, providing the exact input that forward slices to the slice being focused on. Formally, for every $\mathring{t}_a \in \textsc{slices}[a]$ *and* $\mathring{t}_b \in \textsc{slices}[b]$,

$$\underset{aRb}{\mathsf{bwd}}\left(\mathring{t}_b\right) = \mathring{t}_a \text{ if and only if } \mathring{t}_b = \underset{aRb}{\mathsf{fwd}}\left(\mathring{t}_a\right)$$

The definition in Proposition 23 is not always achievable: what if different slices forward slice to the same output? For example, when slicing the single-step trace $1 + 1 \rightsquigarrow 2$, a good choice is to forward slice both the utterly uninformative slice $\blacksquare$ and the minimally informative slice $\blacksquare + \blacksquare$ to the slice $\blacksquare$, as neither input yields any substantive insight about the outcome. In this case, one must choose among different options when backward slicing $\blacksquare$ and a bijection is not possible.

What if forward slicing does not produce all possible slices of the output?[6] How does one determine the appropriate backward slice corresponding to some slice of the output that no input forward slices to? In this case, one would need to choose some slice. This challenge is a blessing in disguise, because addressing this challenge allows programmers to ask questions about any arbitrary slice of the output of a trace they are interested in. Programmers do not have to worry whether their slice of interest is in the image of the forward slicing function or not, which is a formal detail irrelevant to the programmer's intent.

Instead of a bijection that may be unachievable, I specify the minimum requirements for a definition to be acceptable, by defining what counts for a candidate for backward slicing (denoted cbwd):

Definition 15 (A candidate for backward slicing)**.** *A function is a* candidate *for backward slicing if forward slicing its output* $\mathring{t}_a = \underset{aRb}{\mathsf{cbwd}}\left(\mathring{t}_b\right)$*, produces a new slice* $\mathring{t}'_b$ *that preserves at least all the information it started with: in other words, backward-first round trips produce a sound overapproximation. Formally,* [7]

$$\text{For any } \mathring{t}_b \in \textsc{slices}[b],\ \mathring{t}_b \sqsubseteq \underset{aRb}{\mathsf{fwd}}\left(\underset{aRb}{\mathsf{cbwd}}\left(\mathring{t}_b\right)\right)$$

Under this definition, a candidate backward slicing function computes the parts of a that *may* impact a particular selection of the parts of b. Whenever backward slicing cannot be precise, this definition forces it to over approximate relevance. This is useful for debugging purposes, as whatever backward slicing marks as irrelevant *must* not impact the slice of the result the programmer was interested in. Backward slicing should not provide false negatives for relevance: it should never mark parts as irrelevant which could actually be relevant.

But the definition of candidacy does not limit how much extra information backward slicing can reintroduce. The function that never marks anything as irrelevant and always returns a is a valid candidate according to Definition 15. In fact, there might be many candidate functions that fulfill this constraint if it was the only thing required of backward slicing. We need further specification. The purpose for slicing is to help developers who ask questions about trace slices. In this context, the extra information is noise (or imprecision), and I would like to introduce as little noise as possible.

Proposition 24 (Correctness of backward slicing in terms of minimality)**.** *There exists a candidate for backward slicing if and only if there exists one candidate function that is a greatest lower bound (with respect to $\sqsubseteq$) among all candidates, called* bwd*.*

That is, for every other candidate function cbwd*', bwd is smaller for all inputs $\mathring{t}_b$:*

$$\text{For every } \mathsf{cbwd}' \text{ and for every } \mathring{t}_b,\ \underset{aRb}{\mathsf{bwd}'}\left(\mathring{t}_b\right) \sqsubseteq \underset{aRb}{\mathsf{cbwd}'}\left(\mathring{t}_b\right)$$

Proposition 24 establishes that if backward slicing candidates exist, one can choose the candidate that introduces the least extra information. This minimizes the debugging surface for the programmer.

The key contribution of the Galois slicing framework is recognizing that satisfying these requirements is equivalent to requiring that forward and backward slicing form a Galois connection, which weakens the intuitive mirroring requirements of Proposition 23 from an equality constraint to an order preservation constraint:

Proposition 25 (Galois connections satisfy the backward slicing specification)**.** *Suppose* fwd *and* bwd *form a Galois connection, that is, for every $\mathring{t}_a \in \textsc{slices}[a]$ and $\mathring{t}_b \in \textsc{slices}[b]$,*

$$\underset{aRb}{\mathsf{bwd}}\left(\mathring{t}_b\right) \sqsubseteq \mathring{t}_a \text{ if and only if } \mathring{t}_b \sqsubseteq \underset{aRb}{\mathsf{fwd}}\left(\mathring{t}_a\right)$$

Then bwd *is a candidate for backward slicing (Definition 15) and is a minimal candidate (Proposition 24).*

At first sight it may seem one gains nothing: it looks like one not only has to come up with both forward and backward slicing function definitions, but now one also has to prove an extra theorem. But there is an extra gift from the theory of Galois connection: if fwd *distributes over meets*,[8] that is, if for any set of slices it doesn't matter whether one computes their meet and then applies fwd or if one applies fwd first and then computes the meet over all the results, then bwd is uniquely determined: there exists one (and only one) function bwd that forms a Galois connection with fwd and fulfills the criteria from Definition 15 and Proposition 24. By imposing an extra constraint on forward slicing, the workload is diminished: backward slicing needs not be designed from scratch every time. It now gets defined mathematically in terms of the forward slicing function which is easier to define and justify.

Definition 16 (Well behaved forward slicing for meets)**.**
Let $\underset{aRb}{\mathsf{fwd}} : \textsc{slices}[a] \to \textsc{slices}[b]$ be a forward slicing function according to Definition 14, and let $\sqsubseteq$ be a partial order among slices. The forward slicing function is well behaved *for meets if the following conditions hold:*

1. *For any term t, there exist meets or greatest lower bounds among any $\mathring{t}, \mathring{t}' \in \textsc{slices}[t]$ with respect to the partial order $\sqsubseteq$.*

2. fwd *distributes over meets (joins): formally, for all terms a and b such that $a\ R\ b$, and for every set $S \subseteq \textsc{slices}[a]$,*

$$\underset{aRb}{\mathsf{fwd}}\left(\bigsqcap S\right) = \bigsqcap \left\{ \underset{aRb}{\mathsf{fwd}}\left(\mathring{t}_a\right) \;\middle|\; \mathring{t}_a \in S \right\}$$

Forward slicing being well behaved for joins is defined in an analogous fashion.

Theorem 26 (Backward slicing is uniquely determined by well behaved forward slicing)**.** *Let* $\underset{aRb}{\mathsf{fwd}} : \textsc{slices}[a] \to \textsc{slices}[b]$*, be a forward slicing function. If the forward slicing function is well behaved for meets, then backward slicing* bwd *is uniquely determined as follows:*

$$\underset{aRb}{\mathsf{bwd}}\left(\mathring{t}_b\right) = \bigsqcap \left\{ \mathring{t}_a \in \textsc{slices}[a] \;\middle|\; \mathring{t}_b \sqsubseteq \underset{aRb}{\mathsf{fwd}}\left(\mathring{t}_a\right) \right\}$$

4.2.4.1 *Computing backward slicing*

The "well behaved" requirements in Definition 16 suffice to guarantee the existence of a backward slicing function that meets the requirements, but this mathematical definition of bwd does not inspire a particularly practical implementation strategy. In all of the examples in this chapter, computing forward slicing takes linear time over the size of a slice object ($O(a)$). Computing backward slicing from this specification requires checking forward slicing for every possible slice of a, so it takes quadratic time ($O(a^2)$). One would like to also obtain a linear algorithm that specifies backward slicing. This can be achieved if we introduce a few constraints. When the partial order of slices is guaranteed to always have least upper bounds (or *joins*) in addition to having greatest lower bounds (or *meets*), backward slicing can be specified as an algorithm that is easy to compute. All the languages presented in this chapter have both, and as such I present definitions of backward slicing for these languages that, even though uniquely specified, can also be computed as fast as forward slicing, because they can rely on join operations.

4.3 APPLYING THE FRAMEWORK TO THE CARL LANGUAGE

I now dive into the specific details of slicing for CARL. The previous section showed how Galois slicing reduces the definitional burden of a slicing system to only specifying both a partial order and a forward slicing function for each of the relations at hand. By proving some properties about the relationship between them, one can systematically obtain a definition of backward slicing.

I defined slices for CARL in Section 4.2.2, so the only task left is to specify the definition of forward slicing, ensure the candidate definition is well behaved for meets and joins,[9] and reap the benefits of getting a definition of an appropriate well defined backward slicing function. Because the language semantics relies on three relations, three different forward slicing relations require specification, one for each relation in the language: $\rightsquigarrow$, $\longmapsto$, and $\longmapsto^*$. I give details for all of them for CARL. For later languages, most of these developments are reused and I focus on the novel challenges introduced at each step.

4.3.1 *"Notions of forward slicing"* $\left(\underset{t \rightsquigarrow t'}{\mathsf{fwd}}\right)$

CARL has only two notions of reduction, addition and randomness, and we address each separately. For each rule, one can diagram the complete partial order associated with the $\textsc{slices}[t]$ sets, both for the input and the output of a traced reduction. Forward slicing

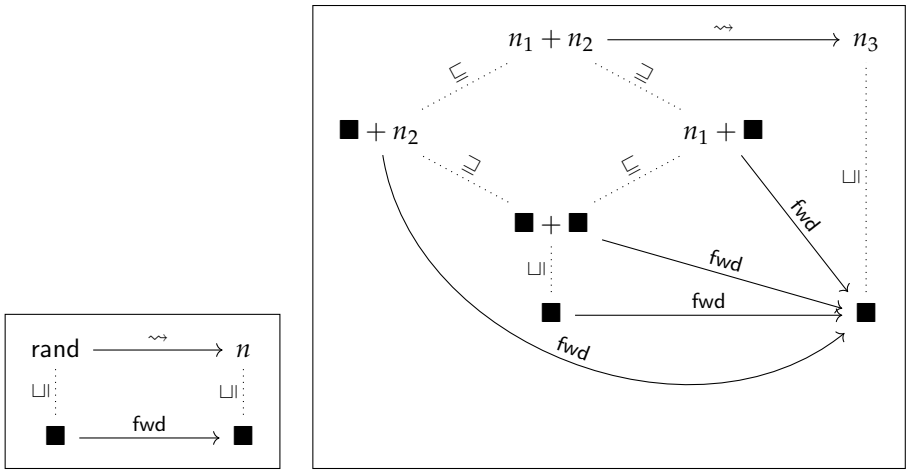


Figure 4.3: Specification of forward slicing for the notions of reduction in the calculator language

is a function from input slices to output slices, and this function can be represented as a collection of arrows in the diagram, one for each of the slices on the set of input slices.

The resulting forward slicing function candidates are presented in Figure 4.3, and I now explain the reasoning behind each arrow. In all cases, the conservative extension restriction from Definition 14 restricts the arrow on the top of the partial order (for the most precise elements), which must reproduce the concrete trace, but the other arrows require further justification.

4.3.1.1 *Forward slicing* rand $\rightsquigarrow n$

The only slice left unspecified for the input partial order is ■, and one must choose to forward slice it either to n or to ■. The Galois slicing formalisms do not reject either choice, as the theorems can accommodate either output slice. We need other means to justify our choice.

The output must also depend on the intuitive meaning that programmers associate to the forward slicing question and to slices themselves. Forward slicing must reflect the slice of the output affected by the slice of the input. The slice ■ must represent a part of the program to be considered as "irrelevant", or "not impacting" the result.

Under this specification for forward slicing, programmers can intuitively expect that marking some code as irrelevant marks its immediate output as irrelevant as well. This criterion forces forward slicing the input slice ■ to the output slice ■. If we chose instead to forward slice ■ to n (in this case, the only other possible choice), we would be saying that the result n is not impacted by an absence of rand in the program. Although the Galois slicing formalism can produce a mathematical development that gives this output, it would go against the programmers intuition and defeat the purpose of forward slicing as an analysis to aid programmers in their debugging tasks. The ■ slice carries the least information and forward slices to ■ in all the languages I discuss in this dissertation.

4.3.1.2 *Forward slicing* $n_1 + n_2 \rightsquigarrow n_3$

The same argument made for rand applies in the context of addition. Forward slicing should not surface information that was hidden or marked "irrelevant" in the input slice. Thus all the other slices should be also sliced down to $\blacksquare$.

This does not mean that one should always forward slice every partially imprecise slice to $\blacksquare$, but is an artifact of the simplicity of the language. If the language had multiplication, where zero is an absorbing element, it would be justified (and compatible with Galois slicing) to forward slice $0 \times \blacksquare$ to 0, for example, when dealing with the trace of $0 \times 5 \rightsquigarrow 0$: when the first multiplicand is 0, multiplication would return 0 no matter the value of the second number. I did not include multiplication in CARL to keep the language simple, but a similar and more relevant situation is discussed in Section 5.3.1.4 when dealing with slicing run-time type errors.

A separate mathematical argument can also justify some of the arrows in forward slicing: some choices do not only break programmer intuitions, but also impede the application of Galois slicing to obtain an induced definition of backward slicing. Forward slicing must distribute over meets to guarantee a unique definition of backward slicing according to Theorem 26. But if one chooses to preserve the result on slices with partial information $\left(\underset{n_1+n_2 \rightsquigarrow n_3}{\mathsf{fwd}} (n_1 + \blacksquare) = \underset{n_1+n_2 \rightsquigarrow n_3}{\mathsf{fwd}} (\blacksquare + n_2) = n_3 \right)$ while forward slicing the less precise slices to $\blacksquare$, forward slicing stops distributing over meets. Consider that the meet between $n_1 + \blacksquare$ and $\blacksquare + n_2$ is $\blacksquare + \blacksquare$. If one chooses to forward slice these slices to n_3, one gets different results when taking the meet before forward slicing than when taking the meet after.

Being careful with the definition of forward slicing also leads to a sensible definition of backward slicing, as one would expect backward slicing n_3 to produce the original program, $n_1 + n_2$. When forward slicing any less precise slice to n_3, backward slicing n_3 would mark some part of the addition as irrelevant, a result that would also break programmer expectations.

The diagrams in Figure 4.3 are quite simple because the output partial order has only two elements, but they also show the case analysis necessary to develop the inference rules for the notions of forward slicing presented in Figure 4.2. In turn, these diagrams uniquely determine the rules representing notions of backward slicing presented in the same figure.

Theorem 27 (Notions of forward slicing for CARL are well behaved for meets and joins)**.** *For any trace* $t_1 \overset{\mathcal{D}}{\rightsquigarrow} t_2$*, the function* $\underset{t_1 \overset{\mathcal{D}}{\rightsquigarrow} t_2}{\mathsf{fwd}}$ *is well behaved for meets and joins (Definition 16).*

By Theorem 26, Theorem 27 implies that there is a unique backward slicing function, which forms a Galois connection with forward slicing. Figure 4.2 provides an alternative direct presentation of backward slicing that produces the same results.

4.3.2 *"Contextual forward slicing"* $\left(\underset{t \longmapsto t'}{\mathit{fwd}}\right)$

A similar process leads to a definition of forward slicing for contextual reduction. Traces have two possible shapes, each relying on some inductive structure: either a derivation is of the shape $\frac{t_1 \rightsquigarrow t_2}{t_1 \longmapsto t_2}$, or a derivation is an inductive step plugged in a particular frame.

For the first case, one can rely on the notions of forward slicing defined previously, since that definition of notions of forward slicing already has all the desired properties:

$$\underset{\frac{t_1 \rightsquigarrow t_2}{t_1 \longmapsto t_2}}{\mathsf{fwd}} \left(\mathring{t}\right) = \underset{t_1 \rightsquigarrow t_2}{\mathsf{fwd}} \left(\mathring{t}\right)$$

The structural reduction rule for plugging a term still needs to be addressed. I first extend the partial order from terms and values to frames, introducing a notion of *frame slices* and their ordering:

$$\frac{\mathring{t} \sqsubseteq \mathring{t}'}{\square + \mathring{t} \sqsubseteq \square + \mathring{t}'} \qquad\qquad \frac{\mathring{v} \sqsubseteq \mathring{v}'}{\mathring{v} + \square \sqsubseteq \mathring{v}' + \square}$$

The goal is to slice a trace of a derivation $F[t_1] \longmapsto F[t_2]$. To achieve this, one must consider all slices of the input, $\mathring{t} \sqsubseteq F[t]$. The following inversion principle guarantees that one accounts for all possible cases:

Lemma 5 (Inversion for slices of a plugged frame)**.** *Let $\mathring{t} \sqsubseteq F[t]$. Then either:*

- $\mathring{t} = \blacksquare$
- *There exists some $\mathring{F} \sqsubseteq F$ and $\mathring{t}' \sqsubseteq t$, such that $\mathring{t} = \mathring{F}[\mathring{t}']$.*

In the particular case of plugging $\blacksquare$ in this language, one could cut the structure of the derivation short as $\blacksquare$ always forward slices to $\blacksquare$. However, for reasons that only become apparent when dealing with mutable state later, I choose instead to depend on the inductive structure and preserve the stack depth of the original derivation, a structure that becomes useful in Section 4.5.3.

Let's turn to the other case of the inversion lemma, $\mathring{t} = \mathring{F}[\mathring{t}']$. Here one can just rely on the inductive structure of the reduction, slicing the plugged slice and then replugging the output in the frame slice, obtaining the specification defined in Figure 4.2. This definition also fulfills the specification for forward slicing.

Theorem 28 (Contextual forward slicing for CARL is well behaved for meets and joins)**.** *For any trace $t_1 \overset{\mathcal{D}}{\longmapsto} t_2$, the function $\underset{t_1 \overset{\mathcal{D}}{\longmapsto} t_2}{\mathit{fwd}}$ is well behaved for meets and joins (Definition 16).*

Again, Theorem 26 uniquely specifies backward slicing. The definition obtained by the specification is equivalent to the set of rules in Figure 4.2.

4.3.3 *Introducing breakpoints on the trace and "Multi-step forward slicing"* $\left(\underset{t \longmapsto^* t'}{\mathit{fwd}}\right)$

I define the reflexive transitive closure of a relation using a common pattern, so I discuss it once, as the same approach works for later languages. Throughout the rest of

Multi-step forward slicing

$$\frac{\dfrac{}{\mathsf{fwd}\ (\mathring{t}) = \mathring{t}}}{t \longmapsto^* t}$$

$$\frac{\dfrac{\mathsf{fwd}_{t \longmapsto t'}\ (\mathring{t}) = \mathring{t}' \qquad \mathsf{fwd}_{t' \longmapsto^* t''}\ (\mathring{t}') = \mathring{t}''}{\mathsf{fwd}_{t \longmapsto t' \quad t' \longmapsto^* t''}\ (\mathring{t}) = \mathring{t}''}}{t \longmapsto^* t''}$$

Multi-step backward slicing

$$\frac{\dfrac{}{\mathsf{bwd}\ (\mathring{t}) = \mathring{t}}}{t \longmapsto^* t}$$

$$\frac{\dfrac{\mathsf{bwd}_{t \longmapsto t'}\ (\mathring{t}') = \mathring{t} \qquad \mathsf{bwd}_{t' \longmapsto^* t''}\ (\mathring{t}'') = \mathring{t}'}{\mathsf{bwd}_{t \longmapsto t' \quad t' \longmapsto^* t''}\ (\mathring{t}'') = \mathring{t}}}{t \longmapsto^* t''}$$

Figure 4.4: Forward and backward slicing multi-step operations

this chapter, I call the reflexive transitive closure of a relation as "the multi-step" version of the relation.

To define breakpoints, I begin from a trace of $m(\geq 0)$ contextual reduction derivation steps:

$$T = t_0 \overset{\mathcal{D}_1}{\longmapsto} \cdots \overset{\mathcal{D}_n}{\longmapsto} t_n \overset{\mathcal{D}_{n+1}}{\longmapsto} \cdots \overset{\mathcal{D}_m}{\longmapsto} t_m$$

To slice this m-step trace, the programmer first chooses an execution step n $(0 \leq n \leq m)$, splitting the trace into two subtraces, $\mathsf{before}_T(n)$ and $\mathsf{after}_T(n)$:

$$\begin{aligned} \mathsf{before}_T(n) &= t_0 \overset{\mathcal{D}_1}{\longmapsto} \cdots \overset{\mathcal{D}_n}{\longmapsto} t_n \\ \mathsf{after}_T(n) &= t_n \overset{\mathcal{D}_{n+1}}{\longmapsto} \cdots \overset{\mathcal{D}_m}{\longmapsto} t_m \end{aligned}$$

The split establishes a breakpoint, and slicing can explore the connection between a slice of the state at the breakpoint step n and any other step on the trace. Forward and backward slicing from any breakpoint still needs the application of slicing to arbitrary derivations of multi-step relations, either forward slicing the after trace or backward slicing the before trace.

A trace is a sequence of single-step contextual reduction derivations, for which forward and backward slicing functions have already been defined. Those functions can be reused to obtain slicing functions for a full trace, which can then be applied to obtain a slicing system that works for any breakpoint chosen by a developer.

One can forward slice derivations of multi-step contextual reduction by simply slicing at each step, and using the result of one step as the input for the next throughout the rest of the trace.

4.3.3.1 *Forward slicing from the beginning of, and backward slicing from the end of, a trace*

How do we reuse the single-step slicing definitions we already have and define with them slicing for multi-step relations? Is composing them sufficient? If Galois connections can be composed into a new Galois connection, we can produce a new multi-step Galois slicing system by combining the single-step Galois slicing systems at each step. Fortunately, this is a general property available for arbitrary Galois connections:

Theorem 29 (Composing Galois connections forms a new Galois connection)**.** *Let A, B, and C be sets with partial orders $\sqsubseteq_A, \sqsubseteq_B$, and $\sqsubseteq_C$, respectively. For any functions $f : A \to B$ and $g : B \to A$ forming a Galois connection with respect to $\sqsubseteq_A$ and $\sqsubseteq_B$, and for any functions $f' : B \to C$ and $g' : C \to B$ forming a Galois connection with respect to $\sqsubseteq_B$ and $\sqsubseteq_C$,* ***the functions*** $(f' \circ f) : A \to C$ ***and*** $(g \circ g') : C \to A$ ***also form a Galois connection (with respect to*** $\sqsubseteq_A$ ***and*** $\sqsubseteq_C$***).***

Since a derivation of a multi-step relation is simply the repeated composition of the contextual reduction relation, this theorem ensures that these derivations can be forward sliced from beginning to end by repeatedly applying single-step forward slicing on the output from the previous step (and the derivation can be backward sliced in the other direction by repeatedly applying single-step backward slicing on the output from the next step). This repeated application for n steps reduces to $n - 1$ compositions of Galois connections. By Theorem 29, this composition also forms a new Galois connection, immediately gaining all the specification requirements for a slicing system. The resulting slicing system is presented in Figure 4.4.

4.3.4 *Minor extension: control flow*

Let's discuss a minor (but interesting) addition to the language, control flow using an if0 construct that branches depending on whether an expression is 0 or some other number. I do not develop this extension in detail, but it serves as an example where the trace itself forces part of the program to be considered irrelevant. Because the trace follows only one branch when reducing an if0 construct, the other branch must always be considered irrelevant for the particular trace.

For example, the semantics of this if0 construct includes the notion of reduction "$(\mathsf{if0}\ 0\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) \rightsquigarrow t_1$", which reduces to the then branch if the predicate is zero. Forward slicing produces a slice of t_1, the outcome of reduction, independent of the slice of t_2 selected in the input. This independence forces that backward slicing always drop the else branch and slice it as ■, otherwise backward slicing would not be minimal.

In general, information discarded during reduction is also discarded during forward slicing, and information discarded during forward slicing has to be marked as irrelevant during backward slicing. This may sound like a formality, but it provides a baseline of parts of a program that are always to be marked as irrelevant by backward slicing. This baseline is consistent with programmer expectations: what is not relevant for the result is marked as ■.

4.4 EXAMPLE 2: ADDING RECURSIVE FUNCTIONS AND SUBSTITUTION

How much extra work is required to provide slicing once interesting features are added to a language? I extend the CARL language with recursive functions to create a new language I call CARF.[10] I do not need to retrace all the steps from the previous section: I add more notions of reduction, but the definitions of contextual reduction and multi-step contextual reduction do not change, so one can reuse their development. Slicing contextual reduction and its multi-step version depend modularly on the notions

Syntax

$$
\begin{array}{rcll}
t \in \textsc{Term} & ::= & \cdots \mid x \mid t\ t & \text{(terms)} \\
v \in \textsc{Value} & ::= & \cdots \mid \mathsf{rec}\ f(x)\,.\,t & \text{(values)} \\
F \in \textsc{Frame} & ::= & \cdots \mid \Box\ t \mid v\ \Box & \text{(frames)}
\end{array}
$$

New notions of reduction

$$
(\mathsf{rec}\ f(x)\,.\,t)\ v \quad \rightsquigarrow \quad [v/x]\,[(\mathsf{rec}\ f(x)\,.\,t)/f]\,t
$$

Figure 4.5: Extensions to the language to include first class functions

of reduction: once slicing is defined for the new terms, one does not need to modify the other parts of the slicing system! Contextual forward and backward slicing can be reused, as well as their multi-step extensions.

Function application in small step semantics is modelled either by using some representation of environments that hold bindings or by using substitution. Previous work used environments to justify slicing systems (Perera, Acar, et al., 2012; Ricciotti et al., 2017). I focus on how to apply Galois slicing to a substitution-based semantics and refer readers interested in environments to the previous work that uses natural semantics.

Figure 4.5 presents the language extensions needed for functions. Of course, the definition of slices must also be extended to accommodate new terms, following the same recipe as earlier:

$$
\begin{array}{rcll}
\mathring{t} \in \textsc{Slice} & ::= & \cdots \mid x \mid \mathring{t}\ \mathring{t} & \text{(slices)} \\
\mathring{v} \in \textsc{VSlice} & ::= & \cdots \mid \mathsf{rec}\ f(x)\,.\,\mathring{t} & \text{(value slices)} \\
\mathring{F} \in \textsc{FSlice} & ::= & \cdots \mid \Box\ \mathring{t} \mid \mathring{v}\ \Box & \text{(frame slices)}
\end{array}
$$

The slice partial order must be extended to with new slices and terms:

$$
\frac{}{x \sqsubseteq x}
\qquad
\frac{\mathring{t}_1 \sqsubseteq \mathring{t}'_1 \qquad \mathring{t}_2 \sqsubseteq \mathring{t}'_2}{\mathring{t}_1\ \mathring{t}_2 \sqsubseteq \mathring{t}'_1\ \mathring{t}'_2}
\qquad
\frac{\mathring{t} \sqsubseteq \mathring{t}'}{\mathsf{rec}\ f(x)\,.\,\mathring{t} \sqsubseteq \mathsf{rec}\ f(x)\,.\,\mathring{t}'}
$$

$$
\frac{\mathring{t} \sqsubseteq \mathring{t}'}{\Box\ \mathring{t} \sqsubseteq \Box\ \mathring{t}'}
\qquad
\frac{\mathring{v} \sqsubseteq \mathring{v}'}{\mathring{v}\ \Box \sqsubseteq \mathring{v}'\ \Box}
$$

4.4.1 *Slicing substitutions*

To slice the extended notions of reduction ($\rightsquigarrow$), I focus on the only new reduction step: function application (β). Function application relies on substitution, so in order to slice applications, forward and backward slicing for substitution has to be defined first. Substitution is a common auxiliary relation in language semantics, and applying the techniques of Galois slicing to substitution facilitates the development of slicing systems for the many language semantics that use it.

First some notation and terminology. I denote substitution by $[t'/x]t$. It replaces all free occurrences of identifier x in the *pre-substitution* term t by the *replacing* term t'.[11] The result is a new term t'', which I call the *post-substitution* term ($t'' = [t'/x]t$).

Substitution is an interesting relation to slice, because substitution is a multiple parameter function. The goal is to provide a slicing system for all traces of any substitution $[t'/x]t = t''$, achieved by defining the following functions, which together must fulfill the standard specification for Galois slicing:

$$\underset{[t'/x]t=t''}{\mathsf{fwd}} : (\textsc{slices}[t'] \times \textsc{slices}[t]) \to \textsc{slices}[t'']$$
$$\underset{[t'/x]t=t''}{\mathsf{bwd}} : \textsc{slices}[t''] \to (\textsc{slices}[t'] \times \textsc{slices}[t])$$

I do not provide for slicing out the identifier position in the substitution, as the effect of slicing out the identifier can be achieved by slicing the replacing term. Slicing out the identifier directly would make the whole substitution slicing more confusing, as the particular trace always depends on the choice of identifier.

Similarly, dealing with pairs in the output makes the formal development hard to read, so I alter the notation a bit to avoid confusion over what part of a substitution each element of a pair represents. First, I introduce two execution functions for backward slicing:

$$\underset{[t'/x]t=t''}{\mathsf{bwdpre}} : \textsc{slices}[t''] \to \textsc{slices}[t] \qquad \underset{[t'/x]t=t''}{\mathsf{bwdrep}} : \textsc{slices}[t''] \to \textsc{slices}[t']$$

These functions represent backward slicing to obtain either a slice of the pre-substitution term (bwdpre) or a slice of the replacing term (bwdrep). This distinction makes it easier to read the inference rules, and it is equivalent to projecting from the standard bwd function. Second, I use an alternative notation for forward slicing of substitutions. The following notational conventions hold:

$$\begin{aligned} \underset{[t'/x]t=t''}{[\mathring{t}'/x]\mathring{t}} &\equiv \underset{[t'/x]t=t''}{\mathsf{fwd}}(\mathring{t}', \mathring{t}) \\ \underset{[t'/x]t=t''}{\mathsf{bwd}}(\mathring{t}'') &\equiv \left(\underset{[t'/x]t=t''}{\mathsf{bwdpre}}(\mathring{t}''), \underset{[t'/x]t=t''}{\mathsf{bwdrep}}(\mathring{t}'') \right) \end{aligned}$$

4.4.1.1 *Specifying forward slicing for substitution*

Forward slicing for substitution is defined in Figure 4.6. Substitution is computed by structural recursion over the pre-substitution term. Forward slicing can follow standard substitution structurally, because term slices replicate the structure of terms. Whenever a ■ slice is in pre-substitution position, forward slicing stops and produces the ■ slice $\left(\underset{[t'/x]t=t''}{[\mathring{t}'/x]\blacksquare} = \blacksquare \right)$.

This definition satisfies the requirements for forward slicing to exist (Definition 14) and to uniquely determine a backward slicing function (Theorem 26):

Theorem 30 (Forward slicing for substitution is well behaved for meets and joins)**.** *For any trace $[t'/x]t = t''$, the function $\underset{[t'/x]t=t''}{[t'/x]t} = t''$ is well behaved for meets and joins (Definition 16).*

$$
\begin{array}{rcl}
\underset{[t'/x]t=t''}{[\mathring{t}'/x]\blacksquare} & = & \blacksquare \\
\underset{[t'/x]n=n}{[\mathring{t}'/x]n} & = & n \\
\underset{[t'/x]\mathsf{rand}=\mathsf{rand}}{[\mathring{t}'/x]\mathsf{rand}} & = & \mathsf{rand} \\
\underset{[t'/x]x=t'}{[\mathring{t}'/x]x} & = & \mathring{t}' \\
\underset{[t'/x]y=y}{[\mathring{t}'/x]y} & = & y \\
 & & \text{when } x \neq y
\end{array}
\qquad
\begin{array}{rcl}
\underset{[t'/x](t_1+t_2)=t_1''+t_2''}{[\mathring{t}'/x](\mathring{t}_1 + \mathring{t}_2)} & = & \underset{[t'/x]t_1=t_1''}{[\mathring{t}'/x]\mathring{t}_1} + \underset{[t'/x]t_2=t_2''}{[\mathring{t}'/x]\mathring{t}_2} \\
\underset{[t'/x]t_1 \mathbin{\mathsf{t}} t_2=t_1''\ t_2''}{[\mathring{t}'/x](\mathring{t}_1 \ \mathring{t}_2)} & = & \underset{[t'/x]t_1=t_1''}{[\mathring{t}'/x]\mathring{t}_1} \ \underset{[t'/x]t_2=t_2''}{[\mathring{t}'/x]\mathring{t}_2} \\
\underset{[t'/x]\mathsf{rec}\ f(x).t=\mathsf{rec}\ f(x).t}{[\mathring{t}'/x]\mathsf{rec}\ f(x) . \mathring{t}} & = & \mathsf{rec}\ f(x) . \mathring{t} \\
\underset{[t'/x]\mathsf{rec}\ f(x).t=\mathsf{rec}\ f(x).t}{[\mathring{t}/f]\mathsf{rec}\ f(x) . \mathring{t}'} & = & \mathsf{rec}\ f(x) . \mathring{t}' \\
\underset{[t'/x]\mathsf{rec}\ f(y).t=\mathsf{rec}\ f(y).t''}{[\mathring{t}/x]\mathsf{rec}\ f(y) . \mathring{t}'} & = & \mathsf{rec}\ f(y) . \underset{[t'/x]t=t''}{[\mathring{t}'/x]\mathring{t}} \\
 & & \text{when } x \neq y \text{ and } x \neq f
\end{array}
$$

Figure 4.6: Forward slicing definition for substitution

4.4.2 *Clarifying the unachievable specification for backward slicing (Proposition 23)*

After introducing substitution, the language becomes interesting enough to more easily see that the bijection mirroring intuition from Proposition 23 is sometimes unachievable: for example, consider the term substitution $[(7+2)/x](x+x) = (7+2)+(7+2)$. When looking for candidates for backward slicing the slice $\mathring{t}' = (\blacksquare + 2) + (7 + \blacksquare)$, the only possibilities for any pre-substitution slice $\mathring{t}$ such $[\mathring{t}_0/x]\mathring{t} = \mathring{t}'$ are either $\blacksquare + \blacksquare$, $x + \blacksquare$, $\blacksquare + x$, or $x + x$ (where in the last one both operands are the same). The closest one could get on the pre-substitution slice would be to keep $x + x$, but in that case, what is to be chosen as a pre-substitution term? None of the options would produce the slice one started from when forward sliced: one must either introduce or lose some information. Any choice that loses information risks hiding some potential bug source for debugging, so we choose instead to over approximate, ensuring that every part not guaranteed to be irrelevant remains in the slice.

4.4.3 *Backward slicing for substitution*

Backward slicing for substitution is uniquely determined. However, it remains useful to take a look at how to extract a linear-time definition of backward slicing, both to confirm that backward slicing does not need to be computed using only the specification in Theorem 26, and to discuss the resulting slices computed by the uniquely determined backward slicing function.

4.4.3.1 *Backward slicing part 1: pre-substitution slice*

To compute the pre-substitution slice from a post-substitution slice, one can recursively follow the structure of the substitution trace as presented in Figure 4.7. Most cases of bwdpre follow the inductive structure of the term, just like a substitution does: the interesting cases arise at the leaves of the derivation. When the trace is cut short by a $\blacksquare$, it stops collecting information. When the trace follows the actual substitution case

$$\frac{}{\underset{[t_0/x]t=t'}{\mathsf{bwdpre}}\ (\blacksquare) = \blacksquare} \qquad \frac{y = f \text{ or } y = x}{\underset{[t_0/y]\mathsf{rec}\ f(x).t=\mathsf{rec}\ f(x).t'}{\mathsf{bwdpre}}\ (\mathsf{rec}\ f(x)\,.\,\mathring{t}') = \mathsf{rec}\ f(x)\,.\,\mathring{t}'} \qquad \frac{}{\underset{[t_0/x]n=n}{\mathsf{bwdpre}}\ (n) = n}$$

$$\frac{y \neq f \text{ and } y \neq x \qquad \underset{[t_0/y]t=t'}{\mathsf{bwdpre}}\ (\mathring{t}') = \mathring{t}}{\underset{[t_0/y]\mathsf{rec}\ f(x).t=\mathsf{rec}\ f(x).t'}{\mathsf{bwdpre}}\ (\mathsf{rec}\ f(x)\,.\,\mathring{t}') = \mathsf{rec}\ f(x)\,.\,\mathring{t}} \qquad \frac{}{\underset{[t_0/x]\mathsf{rand}=\mathsf{rand}}{\mathsf{bwdpre}}\ (\mathsf{rand}) = \mathsf{rand}} \qquad \frac{\mathring{t}' \neq \blacksquare}{\underset{[t'/x]x=t'}{\mathsf{bwdpre}}\ (\mathring{t}') = x}$$

$$\frac{\underset{[t_0/x]t_1=t_1'}{\mathsf{bwdpre}}\ (\mathring{t}_1') = \mathring{t}_1 \qquad \underset{[t_0/x]t_2=t_2'}{\mathsf{bwdpre}}\ (\mathring{t}_2') = \mathring{t}_2}{\underset{[t_0/x]t_1\ t_2=t_1'\ t_2'}{\mathsf{bwdpre}}\ (\mathring{t}_1'\ \mathring{t}_2') = \mathring{t}_1\ \mathring{t}_2} \qquad \frac{\underset{[t_0/x]t_1=t_1'}{\mathsf{bwdpre}}\ (\mathring{t}_1') = \mathring{t}_1 \qquad \underset{[t_0/x]t_2=t_2'}{\mathsf{bwdpre}}\ (\mathring{t}_2') = \mathring{t}_2}{\underset{[t_0/x]t_1+t_2=t_1'+t_2'}{\mathsf{bwdpre}}\ (\mathring{t}_1' + \mathring{t}_2') = \mathring{t}_1 + \mathring{t}_2}$$

$$\frac{x \neq y}{\underset{[t_0/x]y=y}{\mathsf{bwdpre}}\ (y) = y}$$

Figure 4.7: Substitution backward slicing for the pre-substitution term slice

$([t'/x]x = t')$, it inspects the slice $\mathring{t}' \sqsubseteq t'$ at hand: if the slice is not $\blacksquare$, the original identifier x in the trace is relevant, thus it must be kept. Otherwise, the $\blacksquare$ case is triggered. For all other leaf cases, the original term is kept, because it was not directly affected by this substitution.

4.4.3.2 *Backward slicing part 2: replacing slice*

To compute the replacing slice from a post-substitution slice, one can also recursively follow the structure of substitution trace as presented in Figure 4.8. However, the resulting slicing does not follow the pre-substitution term as much as before. Instead, slicing collects only the minimum amount of information about the slicing term actually used in the traced instance of substitution: when the replacing term is not used (e.g. for atomic values), it returns $\blacksquare$.

When multiple premises produce different candidates for the pre-substituted term, it uses joins to combine the information from each. For example, consider the computation of the following replacing slice:

$$\underset{[(1+2)+3/x](x+x)=((1+2)+3)+((1+2)+3)}{\mathsf{bwdpre}}\ ((\blacksquare + 3) + ((1 + \blacksquare) + \blacksquare))$$

In this case, the pre-substituted term is $(x + x)$, and backward slicing for each x produces the replacing term slices $(\blacksquare + 3)$ and $((1 + \blacksquare) + \blacksquare)$, respectively. These two slices are combined by the join (or least upper bound) operation to obtain a slice that contains the information from both slices. In this case, the final resulting replacing slice becomes $((1 + \blacksquare) + 3)$.

Even though joins are not fundamentally required, they let us immediately compute the best answer without having to collect all possible candidates as in the specification definition. The definitions in Figures 4.7 and 4.8 are equivalent to the specification definition of uniquely determined backward slicing.

$$\frac{}{\underset{[t_0/x]t=t'}{\mathsf{bwdrep}}\ (\blacksquare) = \blacksquare} \qquad \frac{y \neq f \text{ and } y \neq x \qquad \underset{[t_0/y]t=t'}{\mathsf{bwdrep}}\ (\mathring{t}') = \mathring{t}_0}{\underset{[t_0/y]\mathsf{rec}\ f(x).t=\mathsf{rec}\ f(x).t'}{\mathsf{bwdrep}}\ (\mathsf{rec}\ f(x)\,.\,\mathring{t}') = \mathring{t}_0} \qquad \frac{y = f \text{ or } y = x}{\underset{[t_0/y]\mathsf{rec}\ f(x).t=\mathsf{rec}\ f(x).t'}{\mathsf{bwdrep}}\ (\mathsf{rec}\ f(x)\,.\,\mathring{t}') = \blacksquare}$$

$$\frac{}{\underset{[t_0/x]n=n}{\mathsf{bwdrep}}\ (n) = \blacksquare} \qquad \frac{}{\underset{[t_0/x]\mathsf{rand}=\mathsf{rand}}{\mathsf{bwdrep}}\ (\mathsf{rand}) = \blacksquare} \qquad \frac{s' \neq \blacksquare}{\underset{[t'/x]x=t'}{\mathsf{bwdrep}}\ (\mathring{t}') = \mathring{t}'} \qquad \frac{x \neq y}{\underset{[t_0/x]y=y}{\mathsf{bwdrep}}\ (y) = \blacksquare}$$

$$\frac{\underset{[t_0/x]t_1=t'_1}{\mathsf{bwdrep}}\ (\mathring{t}'_1) = \mathring{t}_0 a \qquad \underset{[t_0/x]t_2=t'_2}{\mathsf{bwdrep}}\ (\mathring{t}'_2) = \mathring{t}_0 b}{\underset{[t_0/x]t_1\ t_2=t'_1\ t'_2}{\mathsf{bwdrep}}\ (\mathring{t}'_1\ \mathring{t}'_2) = (\mathring{t}_0 a \sqcup \mathring{t}_0 b)} \qquad \frac{\underset{[t_0/x]t_1=t'_1}{\mathsf{bwdrep}}\ (\mathring{t}'_1) = \mathring{t}_0 a \qquad \underset{[t_0/x]t_2=t'_2}{\mathsf{bwdrep}}\ (\mathring{t}'_2) = \mathring{t}_0 b}{\underset{[t_0/x]t_1+t_2=t'_1+t'_2}{\mathsf{bwdrep}}\ (\mathring{t}'_1 + \mathring{t}'_2) = (\mathring{t}_0 a \sqcup \mathring{t}_0 b)}$$

Figure 4.8: Substitution backward slicing for the replacing term slice

This slicing system for substitutions can be used to slice the new notions of reduction in the CARF language.

4.4.4 *Slicing the new notions of reduction in CARF*

Before addressing the recursive function application present in the CARF language, let's analyze a strictly simpler case: function application with non-recursive anonymous functions. Suppose one had instead extended the CARL language with $(\lambda x\,.\,t)\ v \rightsquigarrow [v/x]\,t$ (the standard beta reduction) as a notion of reduction. In this case, one can directly rely on the slicing system for substitution, as substitution is the only interesting step that reduction takes:

$$\frac{\underset{[v/x]t=t''}{[\mathring{v}/x]\mathring{t}\ = \mathring{t}''}}{\underset{(\lambda x.t)\ v\rightsquigarrow[v/x]t}{\mathsf{fwd}}\ ((\lambda x\,.\,\mathring{t})\ \mathring{v}) = \mathring{t}''} \qquad \frac{}{\underset{(\lambda x.t)\ v\rightsquigarrow[v/x]t}{\mathsf{fwd}}\ (\blacksquare\ \mathring{v}) = \blacksquare}$$

$$\frac{}{\underset{(\lambda x.t)\ v\rightsquigarrow[v/x]t}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare}$$

This definition of forward slicing for lambdas is well behaved for meets and joins, thus backward slicing is uniquely determined:

$$\frac{\mathring{t}' \neq \blacksquare \qquad \underset{[v/x]t=t'}{\mathsf{bwdpre}}\ (\mathring{t}') = \mathring{t} \qquad \underset{[v/x]t=t'}{\mathsf{bwdrep}}\ (\mathring{t}') = \mathring{v}}{\underset{(\lambda x.t)\ v\rightsquigarrow[v/x]t}{\mathsf{bwd}}\ (\mathring{t}') = (\lambda x\,.\,\mathring{t})\ \mathring{v}} \qquad \frac{}{\underset{(\lambda x.t)\ v\rightsquigarrow[v/x]t}{\mathsf{bwd}}\ (\blacksquare) = \blacksquare}$$

However, CARF does not have plain lambdas, but rather recursive functions, so reduction must take one step further. Because rec must also bind the recursive function, it performs a double substitution: $(\mathsf{rec}\ f(x)\,.\,t)\ v \rightsquigarrow [v/x]\,[\mathsf{rec}\ f(x)\,.\,t/f]\,t$. Therefore, the

slicing definition relies on two (composed) applications of the substitution slicing. The rules for forward slicing are a bit more complicated given the two substitutions involved:

$$\frac{\underset{[\mathsf{rec}\ f(x)\,.\,t/f]t=t'}{[\mathsf{rec}\ f(x)\,.\,\mathring{t}/f]\mathring{t} = \mathring{t}'} \qquad \underset{[v/x]t'=t''}{[\mathring{v}/x]\mathring{t}' = \mathring{t}''}}{\underset{(\mathsf{rec}\ f(x).t)\ v \rightsquigarrow [v/x][\mathsf{rec}\ f(x).t/f]t}{\mathsf{fwd}}\ \big((\mathsf{rec}\ f(x)\,.\,\mathring{t})\ \mathring{v}\big) = \mathring{t}''}$$

$$\frac{}{\underset{(\mathsf{rec}\ f(x).t)\ v \rightsquigarrow [v/x][\mathsf{rec}\ f(x).t/f]t}{\mathsf{fwd}}\ (\blacksquare\ \mathring{v}) = \blacksquare} \qquad \frac{}{\underset{(\mathsf{rec}\ f(x).t)\ v \rightsquigarrow [v/x][\mathsf{rec}\ f(x).t/f]t}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare}$$

The function body t appears twice in the substitution for f: first in the body of the replacing term, and then as the pre-substitution term. This imposes a more complex definition for backward slicing. Though still uniquely determined, backward slicing must now combine two different premise slices to obtain an output slice for the function. As the trace substitutes first $[\mathsf{rec}\ f(x)\,.\,t/f]t = t'$ and then substitutes the argument x on t', we first backward slice through the later substitution to obtain a slice $\mathring{t}'$ of t'. Then we obtain two slices from the first substitution, a slice $\mathring{t}_f$ of the replacing term $\mathsf{rec}\ f(x)\,.\,t$ and a slice $\mathring{t}_t$ of the pre-substitution term t.

We combine these two slices with a join operation. To ensure that the join is well formed, we wrap the slice $\mathring{t}_t$ of the body of the function with the required syntax to form a slice of the function $\mathsf{rec}\ f(x)\,.\,\mathring{t}_t$. As we have forced one of the slices being joined to have the syntax of a recursive function, the join is guaranteed to always return a recursive function that we can place in the function position when backward slicing the function application.

$$\frac{\underset{[v/x]t'=t''}{\mathsf{bwdrep}}\ (\mathring{t}'') = \mathring{v} \qquad \underset{[v/x]t'=t''}{\mathsf{bwdpre}}\ (\mathring{t}'') = \mathring{t}' \qquad \mathring{t}'' \neq \blacksquare \qquad \underset{[\mathsf{rec}\ f(x).t/f]t=t'}{\mathsf{bwdrep}}\ (\mathring{t}') = \mathring{t}_f \qquad \underset{[\mathsf{rec}\ f(x).t/f]t=t'}{\mathsf{bwdpre}}\ (\mathring{t}') = \mathring{t}_t \qquad \left(\mathring{t}_f \sqcup \mathsf{rec}\ f(x)\,.\,\mathring{t}_t\right) = \mathsf{rec}\ f(x)\,.\,\mathring{t}}{\underset{(\mathsf{rec}\ f(x).t)\ v \rightsquigarrow [v/x][\mathsf{rec}\ f(x).t/f]t}{\mathsf{bwd}}\ (\mathring{t}'') = \big(\mathsf{rec}\ f(x)\,.\,\mathring{t}\big)\ \mathring{v}}$$

$$\frac{}{\underset{(\mathsf{rec}\ f(x).t)\ v \rightsquigarrow [v/x][\mathsf{rec}\ f(x).t/f]t}{\mathsf{bwd}}\ (\blacksquare) = \blacksquare}$$

4.4.5 *Slicing other relations: $\longmapsto$ and $\longmapsto^*$*

Once all notions of reduction are determined, contextual slicing and multi-step slicing should also be addressed. But there are no interesting steps in them: for contextual slicing, the same actions from the CARL language follow, extending to the new frames $\square\ t$, $v\ \square$ and their respective new slices. For multi-step slicing, the whole development was defined in terms of single-step contextual slicing, so the same system from the CARL language can be reused on the CARF language once the slicing systems it depends on have been extended.

4.4.6 *From natural-style slicing to multi-step slicing*

The original presentation of Galois slicing by Perera, Acar, et al. (2012) used an environment-based natural semantics instead of our structural operational approach with substitution. Does this change affect the results of slicing? The short answer is "not in an important way". I leave the formal comparison between both approaches to Appendix C, but here I discuss the relation between both slicing approaches at a high level.

The reduction relations (and therefore the forward slicing functions) in both approaches have different inputs, but these can be made equivalent if we treat environments as value substitutions. Forward slicing an environment and a term for one semantics can be compared to forward slicing the result of substituting that environment into that term for the other semantics.

The reduction relations in both approaches also have different outputs. The environment-based natural semantics produces closure slices that store their current environment, while the small-step approach in this section produces term slices. The resulting slices cannot be equal, but they can be equivalent. Just like the inputs, closure slices can be converted to term slices by substituting the environment into the function body.

The theorems in Appendix C say that once we make the inputs equivalent, we obtain equivalent outputs. The only difference between both forward slicing functions is that environments in the environment-based reduction can contain extra bindings that are never used in the program trace. Given that these bindings are never used, they are discarded when converting closures to terms and not present in the trace of substitution-based reduction. Slicing differences are therefore only "observable" for bindings that remain unused in the program. With the natural semantics, one may still mark these unused bindings as "relevant" in a slicing question, but the answer gains no extra information about them. With the small-step semantics, we can ask questions only about parts used in the computation.

The details required to prove this equivalence are discussed in Appendix C. Among other things, the proof relies on an extra language, a substitution-based natural semantics language and its forward slicing function. This intermediate language simplifies the proof as it separates the equivalence between environments and substitution from the equivalence between natural semantics and small-step operational semantics.

4.5 EXAMPLE 3: ADDING MUTABLE STATE

I now add *mutable state* to the CARF language. In addition to its ubiquity and utility, mutable state exemplifies a language feature that affects the structure of every part of an operational semantics. How much extra non-systematic work is required to produce a slicing system for the language once mutable state is introduced?

The extensions to deal with mutable state are standard and presented in Figure 4.9. I introduce an allocation operation (alloc), a memory access operation (deref), and a memory update operation (set!). I call this language SCARF.[12]

Syntax

$$l \in \textsc{Loc} \quad \mu \in \textsc{Loc} \rightharpoonup \textsc{Value}$$

$$\begin{array}{rcll} t \in \textsc{Term} & ::= & \cdots \mid \mathsf{alloc}\ t \mid \mathsf{deref}\ t \mid \mathsf{set!}\ t\ t & \text{(terms)} \\ v \in \textsc{Value} & ::= & \cdots \mid l & \text{(values)} \\ F \in \textsc{Frame} & ::= & \cdots \mid \mathsf{alloc}\ \Box \mid \mathsf{deref}\ \Box \mid \mathsf{set!}\ \Box\ t \mid \mathsf{set!}\ v\ \Box & \text{(frames)} \end{array}$$

Notions of Reduction (Selection)

$$\begin{array}{rcll} \langle \mathsf{alloc}\ v, \mu \rangle & \rightsquigarrow & \langle l, [l \mapsto v]\mu \rangle & l \notin \mathrm{dom}(\mu) \\ \langle \mathsf{deref}\ l, \mu \rangle & \rightsquigarrow & \langle \mu(l), \mu \rangle & \\ \langle \mathsf{set!}\ l\ v, \mu \rangle & \rightsquigarrow & \langle v, [l \mapsto v]\mu \rangle & l \in \mathrm{dom}(\mu) \end{array}$$

Contextual reduction

$$\frac{\langle t, \mu \rangle \rightsquigarrow \langle t', \mu' \rangle}{\langle t, \mu \rangle \longmapsto \langle t', \mu' \rangle}$$

$$\frac{\langle t, \mu \rangle \longmapsto \langle t', \mu' \rangle}{\langle F[t], \mu \rangle \longmapsto \langle F[t'], \mu' \rangle}$$

Figure 4.9: Extensions to the language to include mutable state

4.5.1 *Extending slices to stores*

At the language syntax level, the process to extend slices to the new operations follows the same systematic approach introduced for CARL, but this is the first language with terms that would never appear in a source program (the explicit memory locations l).[13]

$$\begin{array}{rcll} \mathring{t} \in \textsc{Slice} & ::= & \cdots \mid \mathsf{alloc}\ \mathring{t} \mid \mathsf{deref}\ \mathring{t} \mid \mathsf{set!}\ \mathring{t}\ \mathring{t} & \text{(term slices)} \\ \mathring{v} \in \textsc{VSlice} & ::= & \cdots \mid l & \text{(value slices)} \\ \mathring{F} \in \textsc{FSlice} & ::= & \cdots \mid \mathsf{alloc}\ \Box \mid \mathsf{deref}\ \Box \mid \mathsf{set!}\ \Box\ \mathring{t} \mid \mathsf{set!}\ \mathring{v}\ \Box & \text{(frame slices)} \end{array}$$

The partial order for slices needs to be extended, and again, one can just follow the same structural pattern as before, extending the order to the new term slices, value slices, and frame slices:

$$\frac{}{l \sqsubseteq l} \qquad \frac{\mathring{t} \sqsubseteq \mathring{t}'}{\mathsf{alloc}\ \mathring{t} \sqsubseteq \mathsf{alloc}\ \mathring{t}'} \qquad \frac{\mathring{t} \sqsubseteq \mathring{t}'}{\mathsf{deref}\ \mathring{t} \sqsubseteq \mathsf{deref}\ \mathring{t}'} \qquad \frac{\mathring{t}_1 \sqsubseteq \mathring{t}'_1 \quad \mathring{t}_2 \sqsubseteq \mathring{t}'_2}{\mathsf{set!}\ \mathring{t}_1\ \mathring{t}_2 \sqsubseteq \mathsf{set!}\ \mathring{t}'_1\ \mathring{t}'_2}$$

$$\frac{}{\mathsf{alloc}\ \Box \sqsubseteq \mathsf{alloc}\ \Box} \qquad \frac{}{\mathsf{deref}\ \Box \sqsubseteq \mathsf{deref}\ \Box} \qquad \frac{\mathring{t} \sqsubseteq \mathring{t}'}{\mathsf{set!}\ \Box\ \mathring{t} \sqsubseteq \mathsf{set!}\ \Box\ \mathring{t}'} \qquad \frac{\mathring{v} \sqsubseteq \mathring{v}'}{\mathsf{set!}\ \mathring{v}\ \Box \sqsubseteq \mathsf{set!}\ \mathring{v}'\ \Box}$$

Stores are different from the previously sliced objects as they are not just syntax. Stores are represented by partial functions from locations to values. Thus a *store slice* is also a partial function, but from locations to value slices instead of values. For notation, I use $\mathring{\mu} : \textsc{Loc} \rightharpoonup \textsc{VSlice}$. This definition can be further restricted, because store slices must refer to a particular store:

$$\frac{l \notin \mathrm{dom}(\mu) \qquad l \notin \mathrm{dom}(\mathring{\mu})}{\underset{\langle \mathsf{alloc}\ v,\mu\rangle \rightsquigarrow \langle l,[l\mapsto v]\mu\rangle}{\mathsf{fwd}}(\langle \mathsf{alloc}\ \mathring{v},\mathring{\mu}\rangle) = \langle l,[l\mapsto \mathring{v}]\mathring{\mu}\rangle}$$

$$\frac{l \notin \mathrm{dom}(\mu) \qquad l \notin \mathrm{dom}(\mathring{\mu})}{\underset{\langle \mathsf{alloc}\ v,\mu\rangle \rightsquigarrow \langle l,[l\mapsto v]\mu\rangle}{\mathsf{fwd}}(\langle \blacksquare,\mathring{\mu}\rangle) = \langle \blacksquare,[l\mapsto \blacksquare]\mathring{\mu}\rangle}$$

$$\frac{\mathring{\mu}(l) = \mathring{v} \qquad \mu(l) = v}{\underset{\langle \mathsf{deref}\ l,\mu\rangle \rightsquigarrow \langle v,\mu\rangle}{\mathsf{fwd}}(\langle \mathsf{deref}\ l,\mathring{\mu}\rangle) = \langle \mathring{v},\mathring{\mu}\rangle} \qquad \frac{\mu(l) = v'}{\underset{\langle \mathsf{set!}\ l\ v,\mu\rangle \rightsquigarrow \langle v,[l\mapsto v]\mu\rangle}{\mathsf{fwd}}(\langle \mathsf{set!}\ l\ \mathring{v},\mathring{\mu}\rangle) = \langle \mathring{v},[l\mapsto \mathring{v}]\mathring{\mu}\rangle}$$

$$\frac{\mu(l) = v}{\underset{\langle \mathsf{deref}\ l,\mu\rangle \rightsquigarrow \langle v,\mu\rangle}{\mathsf{fwd}}(\langle \mathsf{deref}\ \blacksquare,\mathring{\mu}\rangle) = \langle \blacksquare,\mathring{\mu}\rangle} \qquad \frac{\mu(l) = v'}{\underset{\langle \mathsf{set!}\ l\ v,\mu\rangle \rightsquigarrow \langle v,[l\mapsto v]\mu\rangle}{\mathsf{fwd}}(\langle \mathsf{set!}\ \blacksquare\ \mathring{v},\mathring{\mu}\rangle) = \langle \blacksquare,[l\mapsto \blacksquare]\mathring{\mu}\rangle}$$

$$\frac{\mu(l) = v}{\underset{\langle \mathsf{deref}\ l,\mu\rangle \rightsquigarrow \langle v,\mu\rangle}{\mathsf{fwd}}(\langle \blacksquare,\mathring{\mu}\rangle) = \langle \blacksquare,\mathring{\mu}\rangle} \qquad \frac{\mu(l) = v'}{\underset{\langle \mathsf{set!}\ l\ v,\mu\rangle \rightsquigarrow \langle v,[l\mapsto v]\mu\rangle}{\mathsf{fwd}}(\langle \blacksquare,\mathring{\mu}\rangle) = \langle \blacksquare,[l\mapsto \blacksquare]\mathring{\mu}\rangle}$$

Figure 4.10: Notions of forward slicing for mutable state operations

Definition 17 (Store slice). *Let $\mu : \textsc{Loc} \rightharpoonup \textsc{Value}$. Then the set of store slices of μ contains all store slices that have the same domain as μ and maps to slices of the values mapped to by μ:*

$$\textsc{slices}[\mu] = \left\{ \mathring{\mu} \in \textsc{Loc} \rightharpoonup \textsc{VSlice} \;\middle|\; \begin{array}{l} dom(\mathring{\mu}) = dom(\mu) \\ \textit{and for each } l \in dom(\mathring{\mu}), \\ \mathring{\mu}(l) \in \textsc{slices}[\mu(l)] \end{array} \right\}$$

Store slices are restricted to have exactly the same domain as the store being sliced. This constraint mirrors the requirements for term slices, where slices must replicate some of the structure of the terms they slice.

4.5.2 *Slicing the notions of reduction relation*

I begin by defining the "notions of forward slicing" relation. All of the notions of reduction that do not depend on the current state just pass around their store slice unmodified, so I focus only on the rules representing the trace of an operation that depends on the store (see Figure 4.10).

The behaviour of forward slicing these notions of reduction is guided by the structure of reduction whenever the slices preserve sufficient information to replicate the original reduction rules. When there is not sufficient information to replicate reduction, we check if the irrelevant parts of the operation impact the store. There are three kinds of operations on the store: reads, writes, and allocations. Whenever a read operation is marked as irrelevant (either by marking just the location as irrelevant or the whole read operation), the operation is treated as if nothing were read: the output term is marked as irrelevant (■), but the store slice remains untouched, because we have not explicitly modified the

store. When a write or an allocation operation is marked as irrelevant (either by marking as irrelevant just the location, just the new value to place at the location, both, or the whole operation), not only the output term is marked as irrelevant, but the content of the particular location in the store slice is also marked as irrelevant, by updating the content of the store at location l to $\blacksquare$. In this way, no information about the contents of the rest of the store is lost, while any future reads to the location are prevented from extracting information that has been now marked as irrelevant.

Does this definition of forward slicing force backward slicing to extremely over approximate the store contents, distracting the programmer with noise while debugging? For example, forward slicing always allocates locations in the store, although some times their contents are marked as $\blacksquare$. Does forward slicing force backward slicing to also keep all allocations in the code, no matter whether their contents are marked as irrelevant? I show why the answer is 'no' when I explain the definition of backward slicing that gets determined.

Theorem 31 (Notions of forward slicing in SCARF are well behaved for meets and joins)**.** *For any trace* $\langle t_1, \mu\rangle \overset{\mathcal{D}}{\rightsquigarrow} \langle t_2, \mu'\rangle$*, the function* $\mathsf{fwd}_{\langle t_1,\mu\rangle \overset{\mathcal{D}}{\rightsquigarrow} \langle t_2,\mu'\rangle}$ *is well behaved for meets and joins (Definition 16).*

4.5.2.1 *Extending substitution slicing*

One must also extend the definition of substitution and substitution slicing. Since the store is not involved in substitution, this extension for this language does not require any new insights. The new language constructs are addressed by following their inductive structure.

4.5.2.2 *Notions of backward slicing*

As forward slicing for the notions of reduction is well behaved for meets and joins, there is a unique definition of backward slicing. In Figure 4.11, I present an equivalent definition that relies on joins to obtain a linear-time backward slicing definition. I use this definition to discuss the impacts of our definition of forward slicing on the information that is preserved by the backward slicing function.

But first, let's discuss what the rules in Figure 4.11 do. Like with forward slicing, these rules not only have to deal with a slice of terms, but they also have to address the contents of the store slice at the relevant locations. All backward slicing rules ending in "-drop" address the case where there is no relevant information both on the slice of the reduction output and on the store location associated with the mutable state operation. The rules with "keep" in their name do not produce $\blacksquare$, instead they preserve some of the structure of the reduction input.

Backward slicing generates slices of the reduction input and of the store "before" a reduction step. In rule [B-deref-keep], the reduction step is reading the contents of the store, and backward slicing must place sufficient information in the store slice at location l to satisfy both the slice of the store contents and the slice of the output. The join operation provides a least upper bound on the information of each slice, so we place that value as

$$\text{B-alloc-keep-l}\ \frac{l \notin \mathrm{dom}(\mu) \qquad l \notin \mathrm{dom}(\mathring{\mu})}{\underset{\langle \mathsf{alloc}\ v,\mu\rangle \rightsquigarrow \langle l,[l\mapsto v]\mu\rangle}{\mathsf{bwd}}(\langle l, [l \mapsto \mathring{v}]\mathring{\mu}\rangle) = \langle \mathsf{alloc}\ \mathring{v}, \mathring{\mu}\rangle}$$

$$\text{B-alloc-keep-v}\ \frac{l \notin \mathrm{dom}(\mu) \qquad \mathring{v} \neq \blacksquare}{\underset{\langle \mathsf{alloc}\ v,\mu\rangle \rightsquigarrow \langle l,[l\mapsto v]\mu\rangle}{\mathsf{bwd}}(\langle \blacksquare, [l \mapsto \mathring{v}]\mathring{\mu}\rangle) = \langle \mathsf{alloc}\ \mathring{v}, \mathring{\mu}\rangle}$$

$$\text{B-alloc-drop}\ \frac{l \notin \mathrm{dom}(\mu) \qquad l \notin \mathrm{dom}(\mathring{\mu})}{\underset{\langle \mathsf{alloc}\ v,\mu\rangle \rightsquigarrow \langle l,[l\mapsto v]\mu\rangle}{\mathsf{bwd}}(\langle \blacksquare, [l \mapsto \blacksquare]\mathring{\mu}\rangle) = \langle \blacksquare, \mathring{\mu}\rangle}$$

$$\text{B-deref-keep}\ \frac{\mathring{\mu}(l) = \mathring{v}' \qquad \mu(l) = v}{\underset{\langle \mathsf{deref}\ l,\mu\rangle \rightsquigarrow \langle v,\mu\rangle}{\mathsf{bwd}}(\langle \mathring{v}, \mathring{\mu}\rangle) = \langle \mathsf{deref}\ l, [l \mapsto \mathring{v} \sqcup \mathring{v}']\mathring{\mu}\rangle}$$

$$\text{B-deref-drop}\ \frac{\mu(l) = v}{\underset{\langle \mathsf{deref}\ l,\mu\rangle \rightsquigarrow \langle v,\mu\rangle}{\mathsf{bwd}}(\langle \blacksquare, \mathring{\mu}\rangle) = \langle \blacksquare, \mathring{\mu}\rangle}$$

$$\text{B-set-keep}\ \frac{\mathring{\mu}(l) = \mathring{v}' \qquad \mu(l) = v' \qquad \mathring{v} \sqcup \mathring{v}' \neq \blacksquare}{\underset{\langle \mathsf{set!}\ l\ v,\mu\rangle \rightsquigarrow \langle v,[l\mapsto v]\mu\rangle}{\mathsf{bwd}}(\langle \mathring{v}, \mathring{\mu}\rangle) = \langle \mathsf{set!}\ l\ \mathring{v} \sqcup \mathring{v}', [l \mapsto \blacksquare]\mathring{\mu}\rangle}$$

$$\text{B-set-drop}\ \frac{\mathring{\mu}(l) = \mathring{v}' \qquad \mu(l) = v' \qquad \mathring{v} \sqcup \mathring{v}' = \blacksquare}{\underset{\langle \mathsf{set!}\ l\ v,\mu\rangle \rightsquigarrow \langle l,[l\mapsto v]\mu\rangle}{\mathsf{bwd}}(\langle \mathring{v}, \mathring{\mu}\rangle) = \langle \blacksquare, [l \mapsto \blacksquare]\mathring{\mu}\rangle}$$

Figure 4.11: Notions of backward slicing for mutable state operations

the contents that the store should have before reduction. Only this way the store slice can gain precision at some location, by collecting the information that is needed by reads. On the other hand, when we update the store as in rule [B-set-keep], we have no information about the previous information in the store, so we "reset" the contents of the store slice at location l to ■. Only this way the store slice can lose precision at some location.

When backward slicing, previous simpler languages could mark the whole reduction input as irrelevant when the output was considered as irrelevant. This approach must be altered when a store is present. If the contents in the store are marked as relevant, backward slicing must keep the operations that populate the store with the relevant store contents. This is why rule [B-alloc-keep-v] produces an allocation operation when backward slicing, even though the location l in the output was marked as irrelevant: its contents are relevant, so the location must be allocated. Memory updates via set! must also make this distinction: rule [B-set-keep] produces a set! operation that would update the store with both the information expected to be present in the store ($\mathring{v}'$) and the parts of v marked as relevant in the output of reduction ($\mathring{v}$). These two are combined with a join operation that generates the least upper bound of slice information.

Although these behaviours are justified locally at a single-step level, they impact the results of backward slicing a trace at a multi-step level. The store slice does not contain only the contents marked by a programmer as relevant, but encodes the information observed by later reads to that location. By the time we reach an allocation, the store slice has accumulated all the information that reads in future steps required, placing that as the slice for the initial value in the [B-alloc-keep-*] rules.

Still, how does this accumulation process interact with backward slicing for set!, which sets the store slice back to ■ at location l? Does this mean that all allocation operations should be preserved, just in case their initial contents are never read before an update? This would introduce considerable noise, and programmers could never trust whether allocation operations are relevant or not to the computation of a result. Fortunately, this is not the case. Let's discuss two example programs, assuming a standard desugaring of `let` as a function application:

```
1 let x := alloc 15 in
2   (set! x 15);
3   15
```

Program a

```
1 let x := alloc 15 in
2   (set! x 15);
3   (deref x)
```

Program b

Both programs produce 15 as a result, but it is reasonable for programmers to expect the slices obtained by backward slicing each program to differ considerably, especially if we only mark the result as relevant and the content of all the locations in the final store as irrelevant. The slicing rules I discussed (after extended to multi-step contextual reduction in the next two subsections) would produce the following slices:

```
1 [redacted]
2 [redacted]
3 15
```

Program a

```
1 let x := alloc [redacted] in
2   (set! x 15);
3   (deref x)
```

Program b

For program a, the store is never accessed, so backward slicing can safely drop both the allocation and the updates. For program b, we only access x after an update, so we do

not care about the contents of the location before line 2. However, we need the allocation operation for x to be marked as relevant, otherwise, where did the location come from? Rule [B-set-keep] keeps a reference to l in line 3, so by the time we backward slice through the substitution for x, the slice for the replacing term must be l, not $\blacksquare$. Thus when we backward slice the first allocation, we can only apply rule [B-alloc-keep-l]. Because the store slice before the set! operation maps l to $\blacksquare$, backward slicing does not preserve the *contents* of the allocation, only the fact that the allocation happened.

4.5.3 *Contextual reduction in the presence of global changes*

To define forward slicing for contextual reduction with mutable state, one can follow the same process described in Section 4.3.2 and summarized in Figure 4.2. Although now one may mark "a value in the store" as something relevant separately from "the expression that calculates its value", the interactions between both arise only in the notions of reduction, so we only need to pass around the stores and rely on the notions of forward slicing to resolve their interactions, defining contextual forward slicing as follows:

$$\frac{\underset{\langle t_1,\mu_1\rangle \rightsquigarrow \langle t_2,\mu_2\rangle}{\mathsf{fwd}}\left(\langle \mathring{t}_1, \mathring{\mu}_1\rangle\right) = \langle \mathring{t}_2, \mu_2\rangle}{\underset{\langle t_1,\mu_1\rangle \longmapsto \langle t_2,\mu_2\rangle}{\mathsf{fwd}}\left(\langle \mathring{t}_1, \mathring{\mu}_1\rangle\right) = \langle \mathring{t}_2, \mu_2\rangle} \qquad \frac{\underset{\langle t_1,\mu_1\rangle \longmapsto \langle t_2,\mu_2\rangle}{\mathsf{fwd}}\left(\langle \mathring{t}_1, \mathring{\mu}_1\rangle\right) = \langle \mathring{t}_2, \mathring{\mu}_2\rangle \quad \mathring{F} \sqsubseteq F}{\underset{\langle F[t_1],\mu_1\rangle \longmapsto \langle F[t_2],\mu_2\rangle}{\mathsf{fwd}}\left(\langle \mathring{F}[\mathring{t}_1], \mathring{\mu}_1\rangle\right) = \langle \mathring{F}[\mathring{t}_2], \mathring{\mu}_2\rangle}$$

$$\frac{\underset{\langle t_1,\mu_1\rangle \longmapsto \langle t_2,\mu_2\rangle}{\mathsf{fwd}}\left(\langle \blacksquare, \mathring{\mu}_1\rangle\right) = \langle \blacksquare, \mathring{\mu}_2\rangle}{\underset{\langle F[t_1],\mu_1\rangle \longmapsto \langle F[t_2],\mu_2\rangle}{\mathsf{fwd}}\left(\langle \blacksquare, \mathring{\mu}_1\rangle\right) = \langle \blacksquare, \mathring{\mu}_2\rangle}$$

Theorem 32 (Contextual forward slicing in SCARF is well behaved for meets and joins)**.** *For any trace $\langle t_1, \mu\rangle \overset{\mathcal{D}}{\longmapsto} \langle t_2, \mu'\rangle$, the function $\underset{\langle t_1,\mu\rangle \overset{\mathcal{D}}{\longmapsto} \langle t_2,\mu'\rangle}{\mathsf{fwd}}$ is well behaved for meets and joins (Definition 16).*

The definition of backward slicing remains uniquely determined by Theorem 26, but our formulation of an optimized linear-time version changes. In the previous languages, forward slicing $\blacksquare$ to $\blacksquare$ determined backward slicing to proceed symmetrically in the structural case. Since backward slicing $\blacksquare$ always produced $\blacksquare$, the structural rule could also drop the frame when backward slicing a trace of $F[t_1] \longmapsto F[t_2]$, as follows:

$$\frac{\underset{t_1 \longmapsto t_2}{\mathsf{fwd}}(\blacksquare) = \blacksquare}{\underset{F[t_1] \longmapsto F[t_2]}{\mathsf{fwd}}(\blacksquare) = \blacksquare} \qquad \frac{\underset{t_1 \longmapsto t_2}{\mathsf{bwd}}(\blacksquare) = \blacksquare}{\underset{F[t_1] \longmapsto F[t_2]}{\mathsf{bwd}}(\blacksquare) = \blacksquare}$$

This symmetry is lost once we have mutable state. Backward slicing the notions of reduction does not always produce $\blacksquare$ from $\blacksquare$, as seen in rules [B-alloc-keep-v] and [B-set-keep] in Figure 4.11.

Because one may mark "the value in the store" as something relevant separately from "the expression that calculates its value", backward slicing may need to reconstruct the minimal expression that still computes the value in the store, even if the whole program was marked as irrelevant.

For example, consider the program "let $x :=$ alloc 10 in (deref $x + 1$)", and a programmer that is only interested in the contents of location l of the store, not the output (in this case, 11). The last step of backward slicing that needs to be computed is

$$\underset{\substack{\langle \mathsf{let}\ x := \mathsf{alloc}\ 10\ \mathsf{in}\ (\mathsf{deref}\ x+1), \cdot\rangle \\ \longmapsto \\ \langle \mathsf{let}\ x := l\ \mathsf{in}\ (\mathsf{deref}\ x+1), [l \mapsto 10]\rangle}}{\mathsf{bwd}} (\langle \blacksquare, [l \mapsto 10]\rangle) = \langle \mathsf{let}\ x := \mathsf{alloc}\ 10\ \mathsf{in}\ \blacksquare), \cdot\rangle$$

The dereferencing and the addition are themselves not important, but the let structure (in our language, a function application) needs to be present so that we can preserve the relevant code, alloc 10, in the right place in the slice.

Contextual backward slicing for traces of the structural rule must distinguish whether the premise has preserved information or not. If the premise does not preserve any information on the program (■), it can follow the same approach as before.

$$\frac{\underset{\langle t_1,\mu\rangle \longmapsto \langle t_2,\mu'\rangle}{\mathsf{bwd}} (\langle \blacksquare, \mathring{\mu}'\rangle) = \langle \blacksquare, \mathring{\mu}\rangle}{\underset{\langle F[t_1],\mu\rangle \longmapsto \langle F[t_2],\mu'\rangle}{\mathsf{bwd}} (\langle \blacksquare, \mathring{\mu}'\rangle) = \langle \blacksquare, \mathring{\mu}'\rangle} \qquad \frac{\underset{\langle t_1,\mu\rangle \longmapsto \langle t_2,\mu'\rangle}{\mathsf{bwd}} (\langle \blacksquare, \mathring{\mu}'\rangle) = \langle \mathring{t}, \mathring{\mu}\rangle \quad \mathring{t} \neq \blacksquare}{\underset{\langle F[t_1],\mu\rangle \longmapsto \langle F[t_2],\mu'\rangle}{\mathsf{bwd}} (\langle \blacksquare, \mathring{\mu}'\rangle) = \ ???}$$

When the premise preserves some term information (as the alloc 10 in our example), backward slicing must be able to keep a slice of the stack, with only the minimal information needed to perform the relevant operation in its original position in the source. To achieve this behaviour I introduce a new function that computes a least frame slice that preserves the topmost syntactic structure of the original frame in the trace and discards the rest of the information in it. I denote this operation $\lfloor F \rfloor$):

$$\begin{array}{rcl@{\qquad}rcl}
\lfloor \square\ t \rfloor & = & \square\ \blacksquare & \lfloor \square + t \rfloor & = & \square + \blacksquare \\
\lfloor v\ \square \rfloor & = & \blacksquare\ \square & \lfloor v + \square \rfloor & = & \blacksquare + \square \\
\lfloor \mathsf{alloc}\ \square \rfloor & = & \mathsf{alloc}\ \square & \lfloor \mathsf{set!}\ \square\ t \rfloor & = & \mathsf{set!}\ \square\ \blacksquare \\
\lfloor \mathsf{deref}\ \square \rfloor & = & \mathsf{deref}\ \square & \lfloor \mathsf{set!}\ v\ \square \rfloor & = & \mathsf{set!}\ \blacksquare\ \square
\end{array}$$

I use this function to deal with the extra case where contextual backward slicing needs to preserve structure. The output backward slice was the result of two least slice computations. The complete set of rules for contextual backward slicing follows:

$$\frac{\underset{\langle t,\mu\rangle \rightsquigarrow \langle t',\mu'\rangle}{\mathsf{bwd}}\left(\langle \mathring{t}',\mathring{\mu}'\rangle\right) = \langle \mathring{t},\mathring{\mu}\rangle}{\underset{\langle t,\mu\rangle \longmapsto \langle t',\mu'\rangle}{\mathsf{bwd}}\left(\langle \mathring{t}',\mathring{\mu}'\rangle\right) = \langle \mathring{t},\mathring{\mu}\rangle} \qquad \frac{\underset{\langle t,\mu\rangle \longmapsto \langle t',\mu'\rangle}{\mathsf{bwd}}\left(\langle \blacksquare,\mathring{\mu}'\rangle\right) = \langle \blacksquare,\mathring{\mu}\rangle}{\underset{\langle F[t],\mu\rangle \longmapsto \langle F[t'],\mu'\rangle}{\mathsf{bwd}}\left(\langle \blacksquare,\mathring{\mu}'\rangle\right) = \langle \blacksquare,\mathring{\mu}\rangle}$$

$$\frac{\underset{\langle t,\mu\rangle \longmapsto \langle t',\mu'\rangle}{\mathsf{bwd}}\left(\langle \mathring{t}',\mathring{\mu}'\rangle\right) = \langle \mathring{t},\mathring{\mu}\rangle \qquad \mathring{F} \sqsubseteq F}{\underset{\langle F[t],\mu\rangle \longmapsto \langle F[t'],\mu'\rangle}{\mathsf{bwd}}\left(\langle \mathring{F}[\mathring{t}'],\mathring{\mu}'\rangle\right) = \langle \mathring{F}[\mathring{t}],\mathring{\mu}\rangle} \qquad \frac{\underset{\langle t,\mu\rangle \longmapsto \langle t',\mu'\rangle}{\mathsf{bwd}}\left(\langle \blacksquare,\mathring{\mu}'\rangle\right) = \langle \mathring{t},\mathring{\mu}\rangle \qquad \mathring{t} \neq \blacksquare}{\underset{\langle F[t],\mu\rangle \longmapsto \langle F[t'],\mu'\rangle}{\mathsf{bwd}}\left(\langle \blacksquare,\mathring{\mu}'\rangle\right) = \langle (\lfloor F \rfloor)\,[\mathring{t}],\mathring{\mu}\rangle}$$

For the first three rules, the only difference with the contextual backward slicing rules for CARL presented in Figure 4.2 is that they include a store, structurally passed to premises until it reaches a derivation for the notions of forward slicing, which requires it, and propagates it unmodified from premises. The fourth rule is new and handles the extra case when backward slicing must preserve some nested operation, as discussed in previous paragraphs.

4.6 EXAMPLE 4: SCARF WITH A REDUCTION SEMANTICS

So far I have used structural operational semantics in every language for its simplicity, but advanced control structures like first class continuations are often captured more easily and clearly using *reduction semantics* (Wright and Felleisen, 1994). In this section, I introduce slicing for a reduction semantics by revisiting the SCARF language from the previous section before extending the language with continuations in Section 4.7. The AGT languages discussed in Chapter 3 are also based on reduction semantics. Developing slicing systems for reduction semantics is thus a required step in providing slicing for AGT languages, which I discuss in Chapter 5.

4.6.1 *Slices of evaluation contexts*

A reduction semantics utilizes a notion of evaluation contexts to represent the runtime evaluation stack, collapsing contextual reduction into a single rule:

$$E \in \textsc{EvCtx} \quad ::= \quad \square \mid F[E] \quad \text{(evaluation context)} \qquad \frac{\langle t,\mu\rangle \rightsquigarrow \langle t',\mu'\rangle}{\langle E[t],\mu\rangle \longmapsto \langle E[t'],\mu'\rangle}$$

I also perform a slight abuse of notation and say that an evaluation context E can be split into two sub contexts E_1 and E_2 using the notation $E = E_1[E_2]$.

Because I did not introduce new notions of reduction, I can reuse the slicing system for notions of reduction from the previous section and focus instead on defining slicing for the reduction semantics itself. Since the novelty is the introduction of evaluation contexts, one must define a way to slice them and a partial order for them:

$$\mathring{E} \in \textsc{ESlice} \quad ::= \quad \square \mid \mathring{F}[\mathring{E}] \quad \text{(evaluation context slices)} \qquad \frac{}{\square \sqsubseteq \square} \qquad \frac{\mathring{F} \sqsubseteq \mathring{F}' \qquad \mathring{E} \sqsubseteq \mathring{E}'}{\mathring{F}[\mathring{E}] \sqsubseteq \mathring{F}'[\mathring{E}']}$$

I have chosen a partial order that preserves the structure of evaluation contexts: when a evaluation context slice loses precision, it does so only on each frame, but it does not lose frames. This approach is a design decision: the goal is to be able to characterize all the terms that are related by the term partial order to an arbitrary term $E[t]$, as these appear in the traces. Terms can lose precision inside-out, thus one must address losing precision on the top of the stack: in particular, slices $E_1[\blacksquare] \sqsubseteq E_1[E_2[t]]$ in the term partial order for every term t and evaluation context E_2. All the possible slices of a term $E[t]$ are guaranteed to be addressed by the following inversion principle, which strongly relies on Lemma 5 and extends its behaviour to full evaluation contexts:

Lemma 6 (Inversion principle for slices of an evaluation context)**.** *Let $\mathring{t} \sqsubseteq E[t]$. Then either:*

- *(Slice preserves stack depth decomposition) There exists some $\mathring{E} \sqsubseteq E$ and $\mathring{t}' \sqsubseteq t$, such that $\mathring{t} = \mathring{E}[\mathring{t}']$*
- *(Irrelevance may drop frames) There exist some partition of the stack into two portions E_1 and E_2 such that $E = E_1[E_2]$, and there exists some $\mathring{E} \sqsubseteq E_1$ such that $\mathring{t} = \mathring{E}[\blacksquare]$.*

One could provide alternative definitions of the partial order for evaluation contexts, e.g., allowing for the removal of frames on the top of the stack, but then one would need to provide a special plug operation for evaluation context slices that accounts for the dropped frames, and from it, a different inversion principle. I choose instead to preserve depth in the partial order and concentrate the loss of precision onto the inversion principle in Lemma 6.

4.6.2 *Forward slicing for reduction semantics*

To provide forward slicing for all valid slices of the terms on the trace in the reduction semantics, I use the inversion principle in Lemma 6. Lemma 6 guarantees that the following two rules address all possible slices of the input terms, and thus provide a complete definition of forward slicing:

$$\frac{\underset{\langle t,\mu\rangle \rightsquigarrow \langle t',\mu'\rangle}{\mathsf{fwd}}(\langle \mathring{t}, \mathring{\mu}\rangle) = \langle \mathring{t}', \mathring{\mu}'\rangle \qquad \mathring{E} \sqsubseteq E \qquad \mathring{t} \sqsubseteq t}{\underset{\langle E[t],\mu \longmapsto E[t'],\mu'\rangle}{\mathsf{fwd}}(\langle \mathring{E}[\mathring{t}]\mathring{\mu}\rangle) = (\langle \mathring{E}[\mathring{t}'], \mathring{\mu}'\rangle)}$$

$$\frac{\underset{\langle t,\mu\rangle \rightsquigarrow \langle t',\mu'\rangle}{\mathsf{fwd}}(\langle \blacksquare, \mathring{\mu}\rangle) = \langle \blacksquare, \mathring{\mu}'\rangle \qquad \mathring{E} \sqsubseteq E_1}{\underset{\langle E_1[E_2[t]],\mu \longmapsto E_1[E_2[t']],\mu'\rangle}{\mathsf{fwd}}(\langle \mathring{E}[\blacksquare]\mathring{\mu}\rangle) = (\langle \mathring{E}[\blacksquare], \mathring{\mu}'\rangle)}$$

Each rule corresponds to one of the possibilities given by the inversion principle in Lemma 6. In the first rule, we have preserved sufficient structure to reach the notion of reduction in the original derivation. In the second rule, we drop some entire term $E_2[t]$ that has been marked as irrelevant, following the "irrelevance may drop frames" portion of the inversion principle. We have no relevant slice of t, so we pass $\blacksquare$ to the notions of forward slicing.

Theorem 33 (Contextual forward slicing in SCARF with a reduction semantics is well behaved for meets and joins)**.** *For any trace* $\langle t_1, \mu\rangle \overset{\mathcal{D}}{\longmapsto} \langle t_2, \mu'\rangle$*, the function* $\underset{\langle t_1,\mu\rangle \overset{\mathcal{D}}{\longmapsto} \langle t_2,\mu'\rangle}{\mathsf{fwd}}$ *is well behaved for meets and joins (Definition 16).*

4.6.3 *Backward slicing for reduction semantics*

As usual, this definition of forward slicing induces a definition of backward slicing. To present this backward slicing as an inductive definition, as I did for the SCARF language, I must consider whether the backward slicing of the notions of reduction from an ■ term slice produces an ■ or not, just like discussed in Section 4.5.3: from the inversion principle for evaluation contexts (Lemma 6), a portion of the stack can be marked as irrelevant (E_2). We can pass the slice ■ to the notions of backward slicing, but this may not necessarily produce ■ as a result. If the notions of forward slicing preserve any information, one needs to reconstruct a "minimal stack", an accumulation of the minimal frames as were needed in the small step development of slicing for SCARF to properly place the slice obtained via the notions of backward slicing. I define minimal stacks by extending the definition of $\lfloor\cdot\rfloor$ to act on evaluation contexts:

$$\lfloor\square\rfloor = \square \qquad \lfloor \mathring{F}[\mathring{E}]\rfloor = \left(\lfloor\mathring{F}\rfloor\right)\left[\left(\lfloor\mathring{E}\rfloor\right)\right]$$

Using this definition, I can characterize the uniquely determined backward slicing function using the following set of rules. When backward slicing a slice that is equivalent to plugging ■, I distinguish two cases depending on whether the output of the notions of backward slicing produces an ■ slice or not. When its output is also an ■ slice, the extra stack frames can be dropped; when it does not, a minimal stack is rebuilt.

$$\frac{\underset{\langle t,\mu\rangle \rightsquigarrow \langle t',\mu'\rangle}{\mathsf{bwd}}\left(\langle \mathring{t}', \mathring{\mu}'\rangle\right) = \langle \mathring{t}, \mathring{\mu}\rangle \qquad \mathring{E} \sqsubseteq E \qquad \mathring{t}' \sqsubseteq t'}{\underset{\langle E[t],\mu \longmapsto E[t'],\mu'\rangle}{\mathsf{bwd}}\left(\langle \mathring{E}[\mathring{t}']\mathring{\mu}'\rangle\right) = \left(\langle \mathring{E}[\mathring{t}], \mathring{\mu}\rangle\right)}$$

$$\frac{\underset{\langle t,\mu\rangle \rightsquigarrow \langle t',\mu'\rangle}{\mathsf{bwd}}\left(\langle \blacksquare, \mathring{\mu}\rangle\right) = \langle \blacksquare, \mathring{\mu}'\rangle \qquad \mathring{E} \sqsubseteq E_1}{\underset{\langle E_1[E_2[t]],\mu \longmapsto E_1[E_2[t']],\mu'\rangle}{\mathsf{bwd}}\left(\langle \mathring{E}[\blacksquare]\mathring{\mu}\rangle\right) = \left(\langle \mathring{E}[\blacksquare], \mathring{\mu}'\rangle\right)}$$

$$\frac{\underset{\langle t,\mu\rangle \rightsquigarrow \langle t',\mu'\rangle}{\mathsf{bwd}}\left(\langle \blacksquare, \mathring{\mu}\rangle\right) = \langle \mathring{t}, \mathring{\mu}'\rangle \qquad \mathring{E} \sqsubseteq E_1 \qquad \mathring{t} \neq \blacksquare}{\underset{\langle E_1[E_2[t]],\mu \longmapsto E_1[E_2[t']],\mu'\rangle}{\mathsf{bwd}}\left(\langle \mathring{E}[\blacksquare]\mathring{\mu}\rangle\right) = \left(\langle \mathring{E}[\left(\lfloor E_2\rfloor\right)\square], \mathring{\mu}'\rangle\right)}$$

This set of rules produces the same slices as the rules presented in Section 4.5.3 for the structural operational semantics version of SCARF.

4.7 EXAMPLE 5: ADDING CONTINUATIONS WITH LET/CC

Now that I have summarized slicing for reduction semantics, I consider the process of slicing first-class continuations. I extend the language as in Figure 4.12, to define the

Syntax

$$
\begin{array}{rcll}
t \in \textsc{Term} & ::= & \cdots \mid \mathsf{let/cc}\ k\ \mathsf{in}\ t \mid \mathsf{throw}\ t\ t & \text{(terms)} \\
v \in \textsc{Value} & ::= & \cdots \mid \mathsf{kont}\ E & \text{(values)} \\
F \in \textsc{Frame} & ::= & \cdots \mid \mathsf{throw}\ \square\ t \mid \mathsf{throw}\ v\ \square & \text{(frames)}
\end{array}
$$

Reduction Semantics

$$
\frac{\langle t, \mu\rangle \rightsquigarrow \langle t', \mu'\rangle}{\langle E[t], \mu\rangle \longmapsto \langle E[t'], \mu'\rangle}
\qquad
\frac{}{\langle E[\mathsf{let/cc}\ k\ \mathsf{in}\ t], \mu\rangle \longmapsto \langle E[[\mathsf{kont}\ E/k]t], \mu\rangle}
$$

$$
\frac{}{\langle E[\mathsf{throw}\ \ (\mathsf{kont}\ E')\ \ v], \mu\rangle \longmapsto \langle E'[v], \mu\rangle}
$$

Figure 4.12: Extensions to the language to include continuations via let/cc

SKARF language,[14] keeping the same notions of reduction ($\rightsquigarrow$) from the SCARF language. I introduce continuations by extending the reduction semantics relation $\longmapsto$, because it is only at this level that one can access the current evaluation context or stack.

Figure 4.12 shows the language extensions for continuations. let/cc binds the current continuation, that is, the evaluation stack of the program, to an identifier by wrapping the evaluation context into a new kind of value, kont E. The continuation can be *thrown* by calling it with a value argument using the throw keyword. When a continuation is thrown, the current evaluation stack is discarded and gets replaced with the evaluation stack that had been previously stored in the continuation.

To slice these new language features, the definition of slices must be extended:

$$
\begin{array}{rcll}
\mathring{t} \in \textsc{Slice} & ::= & \cdots \mid \mathsf{let/cc}\ x\ \mathsf{in}\ \mathring{t} \mid \mathsf{let/cc}\ \blacksquare\ \mathsf{in}\ \mathring{t} \mid \mathsf{throw}\ \mathring{t}\ \mathring{t} & \text{(term slices)} \\
\mathring{v} \in \text{VSlice} & ::= & \cdots \mid \mathsf{kont}\ \mathring{E} & \text{(value slices)} \\
\mathring{F} \in \text{FSlice} & ::= & \cdots \mid \mathsf{throw}\ \square\ \mathring{t} \mid \mathsf{throw}\ \mathring{v}\ \square & \text{(frame slices)}
\end{array}
$$

For let/cc, I not only allow slicing of the body but also of the continuation binding identifier, to distinguish whether the continuation is actually used in the program or not.

One must also extend the partial order for slices to the new slices, and again, one can just follow the same structure as before, extending the order to the new term slices, value slices, and frame slices. I also extend the definition of minimal frames:

$$
\frac{\mathring{t}_1 \sqsubseteq \mathring{t}'_1 \qquad \mathring{t}_2 \sqsubseteq \mathring{t}'_2}{\mathsf{throw}\ \mathring{t}_1\ \mathring{t}_2 \sqsubseteq \mathsf{throw}\ \mathring{t}'_1\ \mathring{t}'_2}
\qquad
\frac{\mathring{t}_k \sqsubseteq \mathring{t}'_k \qquad \mathring{t} \sqsubseteq \mathring{t}'}{\mathsf{let/cc}\ \mathring{t}_k\ \mathsf{in}\ \mathring{t} \sqsubseteq \mathsf{let/cc}\ \mathring{t}'_k\ \mathsf{in}\ \mathring{t}'}
\qquad
\frac{\mathring{t} \sqsubseteq \mathring{t}'}{\mathsf{throw}\ \square\ \mathring{t} \sqsubseteq \mathsf{throw}\ \square\ \mathring{t}'}
$$

$$
\frac{\mathring{t}_v \sqsubseteq \mathring{t}'_v}{\mathsf{throw}\ \mathring{t}_v\ \square \sqsubseteq \mathsf{throw}\ \mathring{t}'_v\ \square}
\qquad
\begin{array}{rcl}
\lfloor \mathsf{throw}\ \square\ t \rfloor & = & \mathsf{throw}\ \square\ \blacksquare \\
\lfloor \mathsf{throw}\ v\ \square \rfloor & = & \mathsf{throw}\ \blacksquare\ \square
\end{array}
$$

I discuss forward slicing and the inductive definition of backward slicing separately for each reduction rule in the semantics: first for the reduction of the throw construct and then for the reduction of the let/cc construct. Notice that the definition of each one follows the inversion principle for evaluation contexts.

Figure 4.13 shows the slicing rules for throw. For forward slicing, the first rule structurally replicates the flow of reduction, as the slice has kept sufficient partial information

$$\frac{}{\underset{\langle E[\mathsf{throw}\ (\mathsf{kont}\ E')\ v],\mu\rangle \longmapsto \langle E'[v],\mu\rangle}{\mathsf{fwd}}\ \left(\langle \mathring{E}[\mathsf{throw}\ \left(\mathsf{kont}\ \mathring{E}'\right)\ \mathring{v}],\mathring{\mu}\rangle\right) = \left(\langle \mathring{E}'[\mathring{v}],\mathring{\mu}\rangle\right)}$$

$$\frac{}{\underset{\langle E[\mathsf{throw}\ (\mathsf{kont}\ E')\ v],\mu\rangle \longmapsto \langle E'[v],\mu\rangle}{\mathsf{fwd}}\ \left(\langle \mathring{E}[\mathsf{throw}\ \blacksquare\ \mathring{v}],\mathring{\mu}\rangle\right) = \left(\langle \blacksquare,\mathring{\mu}\rangle\right)}$$

$$\frac{\mathring{E} \sqsubseteq E_1}{\underset{\langle E_1[E_2[\mathsf{throw}\ (\mathsf{kont}\ E')\ v]],\mu\rangle \longmapsto \langle E'[v],\mu\rangle}{\mathsf{fwd}}\ \left(\langle \mathring{E}[\blacksquare],\mathring{\mu}\rangle\right) = \left(\langle \blacksquare,\mathring{\mu}\rangle\right)}$$

$$\frac{\mathring{E}' \sqsubseteq E' \qquad \mathring{E}'[\mathring{v}] \neq \blacksquare}{\underset{\langle E[\mathsf{throw}\ (\mathsf{kont}\ E')\ v],\mu\rangle \longmapsto \langle E'[v],\mu\rangle}{\mathsf{bwd}}\ \left(\langle \mathring{E}'[\mathring{v}],\mathring{\mu}\rangle\right) = \left(\langle (\lfloor E \rfloor)\,[\mathsf{throw}\ \left(\mathsf{kont}\ \mathring{E}'\right)\ \mathring{v}],\mathring{\mu}\rangle\right)}$$

$$\frac{\mathring{E}' \sqsubseteq E_1' \qquad \mathring{E}'[\blacksquare] \neq \blacksquare}{\underset{\langle E[\mathsf{throw}\ \left(\mathsf{kont}\ E_1'[E_2']\right)\ v],\mu\rangle \longmapsto \langle E_1'[E_2'[v]],\mu\rangle}{\mathsf{bwd}}\ \left(\langle \mathring{E}'[\blacksquare],\mathring{\mu}\rangle\right) = \left(\langle (\lfloor E \rfloor)\,[\mathsf{throw}\ \left(\mathsf{kont}\ \mathring{E}'[(\lfloor E_2 \rfloor)]\right)\ \blacksquare],\mathring{\mu}\rangle\right)}$$

$$\frac{}{\underset{\langle E[\mathsf{throw}\ (\mathsf{kont}\ (E_1[E_2]))\ v],\mu\rangle \longmapsto \langle E_1'[E_2'[v]],\mu\rangle}{\mathsf{bwd}}\ \left(\langle \blacksquare,\mathring{\mu}\rangle\right) = \left(\langle (\lfloor E \rfloor)\,\blacksquare,\mathring{\mu}\rangle\right)}$$

Figure 4.13: Slicing rules for throw

from the continuation that is being thrown. Whenever the thrown continuation is marked as irrelevant, there is no information about the control stack after the throw, thus forward slicing produces ■ on the term side.

For backward slicing, we follow the inversion principle for evaluation contexts to reconstruct the relevant portion of the evaluation context stored in the kont value. When part of the stack has been sliced out, the rest of the structure is reconstructed with the "minimal stack" reconstruction technique discussed in Section 4.6.3. When backward slicing a continuation throw, one not only needs to rebuild the stack that came from the continuation, but the evaluation context that was present *before* the continuation throw. Since the trace at this point only cares that the continuation throw position is reached, we rebuild only the minimal stack $\lfloor E \rfloor$ for the pre-throw evaluation context in every backward slicing rule.

Figure 4.14 shows the slicing rules for let/cc. In forward slicing, slicing the binder out (as in let/cc ■ in $\cdots$) is interpreted as considering the continuation identifier k as irrelevant, so its occurrences can be substituted with the replacing slice ■. If the identifier is marked as relevant instead, occurrences of k are substituted with the current evaluation context slice, in a process that mimics the original behaviour of let/cc. If the whole let/cc is marked as irrelevant, or if part of the stack is dropped via ■, then forward slicing just preserves the (little) information it had available.

The rules for backward slicing are equivalent to the uniquely determined definition. They rely on extensions of substitution slicing to account for continuations. These extensions follow the same intuitions as I have discussed in previous sections for function

$$\frac{\underset{[\mathsf{kont}\ E/k]t=t'}{[\mathsf{kont}\ \mathring{E}/k]\mathring{t}\ =\ \mathring{t}'}}{\underset{\langle E[\mathsf{let/cc}\ k\ \mathsf{in}\ t],\mu\rangle\longmapsto\langle E[[\mathsf{kont}\ E/k]t],\mu\rangle}{\mathsf{fwd}}\ \left(\langle\mathring{E}[\mathsf{let/cc}\ k\ \mathsf{in}\ \mathring{t}],\mathring{\mu}\rangle\right)=\left(\langle\mathring{E}[\mathring{t}'],\mathring{\mu}\rangle\right)}$$

$$\frac{\underset{[\mathsf{kont}\ E/k]t=t'}{[\blacksquare/k]\mathring{t}\ \ =\ \mathring{t}'}}{\underset{\langle E[\mathsf{let/cc}\ k\ \mathsf{in}\ t],\mu\rangle\longmapsto\langle E[[\mathsf{kont}\ E/k]t],\mu\rangle}{\mathsf{fwd}}\ \left(\langle\mathring{E}[\mathsf{let/cc}\ \blacksquare\ \mathsf{in}\ \mathring{t}],\mathring{\mu}\rangle\right)=\left(\langle\mathring{E}[\mathring{t}'],\mathring{\mu}\rangle\right)}$$

$$\frac{\mathring{E}\sqsubseteq E_1}{\underset{\langle E_1[E_2[\mathsf{let/cc}\ k\ \mathsf{in}\ t]],\mu\rangle\longmapsto\langle E_1[E_2[[\mathsf{kont}\ E_1[E_2]/k]t]],\mu\rangle}{\mathsf{fwd}}\ \left(\langle\mathring{E}[\blacksquare],\mathring{\mu}\rangle\right)=\left(\langle\mathring{E}[\blacksquare],\mathring{\mu}\rangle\right)}$$

$$\frac{\underset{[\mathsf{kont}\ E/k]\mathring{t}=t'}{\mathsf{bwdpre}}\ \left(\mathring{t}'\right)=\mathring{t}\qquad\underset{[\mathsf{kont}\ E/k]\mathring{t}=t'}{\mathsf{bwdrep}}\ \left(\mathring{t}'\right)=\mathsf{kont}\ \mathring{E}'\qquad\mathring{t}\neq\blacksquare}{\underset{\langle E[\mathsf{let/cc}\ k\ \mathsf{in}\ t],\mu\rangle\longmapsto\langle E[[\mathsf{kont}\ E/k]t],\mu\rangle}{\mathsf{bwd}}\ \left(\langle\mathring{E}[\mathring{t}'],\mathring{\mu}\rangle\right)=\left(\langle\left(\mathring{E}\sqcup\mathring{E}'\right)[\mathsf{let/cc}\ k\ \mathsf{in}\ \mathring{t}],\mathring{\mu}\rangle\right)}$$

$$\frac{\underset{[\mathsf{kont}\ E/k]\mathring{t}=t'}{\mathsf{bwdpre}}\ \left(\mathring{t}'\right)=\mathring{t}\qquad\underset{[\mathsf{kont}\ E/k]\mathring{t}=t'}{\mathsf{bwdrep}}\ \left(\mathring{t}'\right)=\blacksquare\qquad\mathring{t}\neq\blacksquare}{\underset{\langle E[\mathsf{let/cc}\ k\ \mathsf{in}\ t],\mu\rangle\longmapsto\langle E[[\mathsf{kont}\ E/k]t],\mu\rangle}{\mathsf{bwd}}\ \left(\langle\mathring{E}[\mathring{t}'],\mathring{\mu}\rangle\right)=\left(\langle\mathring{E}[\mathsf{let/cc}\ \blacksquare\ \mathsf{in}\ \mathring{t}],\mathring{\mu}\rangle\right)}$$

$$\frac{\underset{[\mathsf{kont}\ E/k]\mathring{t}=t'}{\mathsf{bwdpre}}\ \left(\mathring{t}'\right)=\blacksquare}{\underset{\langle E[\mathsf{let/cc}\ k\ \mathsf{in}\ t],\mu\rangle\longmapsto\langle E[[\mathsf{kont}\ E/k]t],\mu\rangle}{\mathsf{bwd}}\ \left(\langle\mathring{E}[\mathring{t}'],\mathring{\mu}\rangle\right)=\left(\langle\mathring{E}[\blacksquare],\mathring{\mu}\rangle\right)}$$

$$\frac{\mathring{E}\sqsubseteq E_1}{\underset{\langle E_1[E_2[\mathsf{let/cc}\ k\ \mathsf{in}\ t]],\mu\rangle\longmapsto\langle E_1[E_2[[\mathsf{kont}\ E/k]t]],\mu\rangle}{\mathsf{bwd}}\ \left(\langle\mathring{E}[\blacksquare],\mathring{\mu}\rangle\right)=\left(\langle\mathring{E}[\blacksquare],\mathring{\mu}\rangle\right)}$$

Figure 4.14: Slicing rules for let/cc

application: there is only one substitution in the reduction of let/cc, so we can recover the body of the let/cc from backward slicing the substitution with bwdpre, as well as a relevant slice of the previous evaluation context from backward slicing the substitution with bwdrep. When this slice of the evaluation context is not ■, its information must be combined with the slice of the evaluation context with a join, as seen in the first rule. In the other rules, the join is omitted as it is always redundant.

Theorem 34 (Contextual forward slicing in SKARF with a reduction semantics is well behaved for meets and joins)**.** *For any trace* $\langle t_1, \mu \rangle \overset{\mathcal{D}}{\longmapsto} \langle t_2, \mu' \rangle$*, the function* $\mathsf{fwd}_{\langle t_1,\mu\rangle \overset{\mathcal{D}}{\longmapsto} \langle t_2,\mu'\rangle}$ *is well behaved for meets and joins (Definition 16).*

4.8 CONCLUSION

In this chapter I have shown how to use Galois slicing by example, providing a slicing system for a set of increasingly advanced languages. The key advantage of Galois slicing is that it provides a systematic approach to obtain a forward slicing system for traces, while specifying a backward slicing system in terms of the forward slicing definition. This approach reduces the amount of design work that a semantics engineer must perform to obtain a slicing system. The different languages I have explored show that Galois slicing scales to advanced programming language features.

I believe this by-example guide can help semantics engineers apply Galois slicing to many other language features. Semantics engineers and tool builders can then prove debugging and exploration tools for their languages based on these foundations. This chapter also provides detailed formal background needed to develop Galois slicing tools for AGT gradual languages, the topic I pursue in the next chapter.

4.9 NOTES

1. CARL stands for "Calculator with Randomness Language".
2. I have simplified the trace to only show the complete terms as they step. The way I approach Galois slicing treats derivation trees as essential, representing the structural trace of a step of evaluation. I require that each step includes a derivation in the contextual reduction relation.
3. The original work used transparent boxes ($\square$) to represent irrelevant sub-expressions. However, that notation collides with the notation uses for holes in reduction semantics and in AGT. To avoid confusion, in this thesis I choose to use $\blacksquare$ instead.
4. Once mutable state is introduced, stores will also need slicing. See Section 4.5.
5. In the Galois slicing literature, this set is usually called $\downarrow t$.
6. Formally, we would say in that case that forward slicing is not a surjective/onto function.
7. Ricciotti et al. (2017) call this formal property "consistency".
8. The formal literature would say that fwd "preserves" meets. A *meet* is another name for greatest lower bound. A *join* is another name for least upper bound.
9. If the candidate definition is not well behaved for meets and joins the designer needs to iterate, refine their forward slicing design, and try again.
10. CARF stands for CArl with Recursive Functions.
11. I took the name "replacing term" from Steele (2017).
12. SCARF stands for State CARF.
13. It is traditional in PL theory developments to conflate run-time program configurations with terms and to consider any kind of computation as term rewriting.
14. The name change from SCARF to SKARF is intended to evoke continuations, which are traditionally represented with the letter k.

Although a language can never teach a programmer what constitutes a well-designed program, it can guide him into thinking about the right things.

— Liskov and Zilles (1974)

5 AGT THAT EXPLAINS ITS FAILURES

Run-time type errors happen in the languages designed using the AGT framework. The designs from Chapter 3 abort execution whenever a type inconsistency is found at run time, without providing information about the source of the problem at hand.

I increase the information available to programmers when a run time type error happens by introducing slicing systems. A slicing system allows programmers to distinguish the parts of a program involved in a computation from those that are not, both for successful computations and for run-time type failures. Providing a slicing system for an AGT language helps programmers identify the parts of the program that led to the type inconsistency that triggered a particular run-time type error.

A slicing system can be used beyond run-time type errors. It may also be used by programmers to refine their understanding of a program's behaviour in general and to ask other debugging questions over a program execution trace.

In Chapter 4, I discuss Galois slicing, which provides a systematic approach to the design of slicing systems for programming languages. The key contribution of this chapter is applying Galois slicing to obtain slicing systems for AGT-based languages, providing systematic design guidelines that can be applied to future language designs.

I revisit two languages discussed in Chapter 3 and introduce slicing systems for them using Galois slicing. The first language is AGTLC, a simple language where I can focus on the main ideas and structure of Galois slicing for AGT languages. I then discuss the extensions required to provide a slicing system for GTFL$_{\lesssim}$ with Bounded Rows and Records.

5.1 SLICING AS ERROR REPORTING

The original semantics of an AGT language identifies type inconsistencies at run time and aborts execution, but that's it. The program stops without further detail. The original semantics of AGT languages provides no indication of where in the program does the inconsistency come from.

A run-time type failure may signal multiple simultaneous inconsistencies, each traceable to different portions of a program. For example, consider the following program,

which I designed to fail with a run-time type error, as the *input* value is incompatible with the type of *config*:

$$\begin{array}{l}\mathsf{let}\ \mathit{input} : ? := [\mathit{transform} : 2, \mathit{size} : \lambda(x : \mathsf{Int}).x]\ \mathsf{in} \\ \quad \mathsf{let}\ \mathit{config} : [\mathit{size} : \mathsf{Int}, \mathit{transform} : \mathsf{Int} \to \mathsf{Int}] := \mathit{input}\ \mathsf{in} \\ \quad\quad (\mathit{config}.\mathit{transform})\ (\mathit{config}.\mathit{size})\end{array}$$

I swapped the mappings of the fields of the record to simulate a programming mistake, thus both fields have incorrect types. Because *input* is declared as ?, this inconsistency is not detected when the program is compiled, but the runtime of the language finds the inconsistency and stops execution.

Because I designed the program to fail, I could reveal the error I injected in the program. *I*, as this program's designer, know the problem is the swap, but the semantics cannot assume this is the problem in this program. To the semantics it is the same whether the fields were swapped, whether the record is correct but a separate *input*2 value in the context should have been used, whether the value for *input* is correct but the type annotation for *config* is incorrect (and thus the uses on the body are also incorrect), or whether some other situation leads the program to have these inconsistent types. As motivated in Chapters 1 and 2, this is external knowledge that the programmer has and is not encoded in the program. Only the programmer can judge what needs to change in a program, not the language. But the language can still see an inconsistency among type constraints and must stop execution. I believe it would help developers to report these constraints.

My goal is to provide a system that the programmer can use to disregard parts of the program guaranteed to not be involved in a run-time type failure. This system shows inconsistencies, but that is as far as the language can go: the judge of culpability is the programmer. But for the programmer to be able to see the swap issue in this program, they cannot be shown partial information on the error. For example, reporting partial information like "the field *size* in *config* has type $\mathsf{Int} \to \mathsf{Int}$ when it was declared to be Int" omits what happens with *transform* and may lead the programmer on a (perhaps unsuccessful) search to fix only part of the inconsistency. When that problem is fixed in isolation, the program remains inconsistent. The information provided must refer to both problems.

I go beyond simply having errors return diagnostic messages; they instead provide *program slices*. Program slices are a debugging tool that provides sufficient information to identify all the inconsistencies in a program.[1]

But what is "sufficient information"? This is not sufficiently precise, but we can progressively refine the definition of "sufficient":

- **"Sufficient" is an upper bound, but that could possibly mean "The whole program"** A first definition of "Sufficient information" is to provide an upper bound on the information related to a run-time error. But this definition does not require a language to discard any information, and therefore it would accept the original AGT semantics, which provides no details: If we take "sufficient" as just providing an upper bound on the information, we could just say "There are inconsistencies in this program" and satisfy the "sufficient information" constraint. I do not want this.

I thus cannot use this definition to improve on the error reporting, as it does not provide further restrictions than the requirements imposed by type safety. I want a more precise definition, distinguishing parts of the program not involved in a failure from those involved.

- **Gradual typing is designed to hide some errors.** A gradual program may hold many latent inconsistencies that are not involved in a particular failure: they may either arise later, depending on some other problem in the program, or may not arise if they would manifest only in a path outside of an execution's control flow. I do not want "sufficient information" to include *all possible* inconsistencies in a program, but the inconsistencies that have been detected in an execution trace and that cause a particular run-time error.

- **I intend to improve error information, not introduce or remove errors**. Presenting *all* inconsistencies in a program and stopping on *any* inconsistency, including those not observed when the program was run, would require altering the language semantics. The key advantage of gradual typing is that some type inconsistencies may lurk in a program, but if they are not on the control flow of a particular execution, they do not trigger a failure. Presenting all possible inconsistencies may distract from the cause of a particular failure, and stopping in the presence of any inconsistency would alter the semantics of the language. I want a system that does not change whether a program produces an error or not, but rather focuses on improving the information available in the case of existing failures.

5.1.1 *Galois slicing provides developers with feedback*

Program slicing is a technique that provides sufficient information to reproduce a failure but is also minimal, introducing as little irrelevant information as possible, helping developers identify parts of a program *not involved* in a failure.

I use Galois slicing to provide debugging tools that satisfy this definition of sufficient information. I discussed the general design of a slicing system using Galois slicing in Chapter 4. If one designs a forward slicing system for a language, a system allowing users to distinguish parts involved in a computation or failure, then Galois slicing provides a backward slicing function that provides the smallest portion of a program that reproduces a failure. Backward slicing a run-time type failure provides information distinguishing parts of the program not involved in the failure.

I design a slicing system that, when backward slicing from the error, provides the following slice from the execution trace of the previously discussed example program:

$$\begin{array}{l} \mathsf{let}\ \mathit{input} : \blacksquare := [\mathit{transform} : 2, \mathit{size} : \lambda(x : \blacksquare).x]\ \mathsf{in} \\ \quad \mathsf{let}\ \mathit{config} : [\mathit{size} : \mathsf{Int}, \mathit{transform} : \blacksquare \to \blacksquare] := \mathit{input}\ \mathsf{in} \\ \qquad \blacksquare \end{array}$$

I describe the slicing process that leads to this result in Section 5.6.3, after introducing Galois slicing for the languages presented in Chapter 3.

5.2 SLICING AGT AT A HIGH LEVEL

A slicing system provides forward and backward slicing functions that can answer programmers' questions: "what portion/slice of some later state is affected by the slice of this state" (forward slicing) or "what portion/slice of some earlier state affects the slice of this state" (backward slicing). These questions may be asked at any intermediate point of an evaluation trace, which can be considered as an evaluation breakpoint when dealing with small-step and reduction semantics.

I describe program slicing at a high level in Section 4.2, but there are some differences in the process of slicing an AGT language. Slicing for an AGT language requires combining slicing systems for the multiple relations involved in an AGT semantics. From the source program until a run-time type error arises, a program passes through various steps and relations: it is first elaborated into an intermediate language, and the elaboration process depends on the "initial evidence" operation to generate a run-time representation of the assumptions made by the static type system. Then the program is run in the intermediate language. The reduction semantics propagates run-time type information using an "evidence composition" operation. When one of these operations fails, a run-time type error is triggered.

Designing a slicing system for an AGT language requires dealing with each of these relations and operations and consists at a high level of the following steps:

- Ensure all partial functions are turned into *total functions* by making the failure state explicit. This process makes the design of the slicing systems more uniform.
- Design a slicing system for initial evidence.
- Use the slicing system for initial evidence to design a slicing system for the elaboration algorithm that connects source programs to their runtime representation.
- Design a slicing system for evidence composition. This may be done by relying on the inductive structure of evidence composition when available (as I do here for AGTLC) or by relying on the equivalence (when available) between evidence composition and a combination of gradual meet and initial evidence operations, which can reuse the slicing system for initial evidence (as I do in Section 5.4 for $\text{GTFL}_{\lesssim}$ with BRR).
- Use the slicing system for evidence composition to design a slicing system for the notions of reduction.
- Use both the slicing system for evidence composition and the slicing system for the notions of reduction to design a slicing system for contextual reduction.
- Using the slicing system for contextual reduction, one can generate a slicing system for multi-step reduction traces using the techniques described in Section 4.3.3.

The core extra steps to provide slicing systems for AGT are the slicing subsystems for evidence: a slicing system for the calculation of initial evidence, a slicing system for the elaboration algorithm that introduces evidence into a program, and a slicing system

for evidence composition. Once these three subsystems are done, the rest of the work to produce a complete slicing system for an AGT language reduces to providing slicing for a substitution-based reduction semantics, which I discuss in Section 4.6.

I discuss these slicing subsystems in the simple context of AGTLC first, and later, I discuss the extra requirements introduced by a more advanced type discipline, as in the case of Bounded Rows and Records. Both languages are introduced in Chapter 3. Although AGTLC is a simple language, its complexity is comparable to the languages where most of the work on alternative error reporting approaches for gradual typing has been developed (blame, as discussed in Sections 2.2.3 and 2.5.1).

5.3 SLICING ELABORATION FOR AGTLC

The first step in providing a slicing system for an AGT language is to provide a slicing system for elaboration. I present the elaboration algorithm when introducing AGTLC in Section 3.1.8. This algorithm transforms well typed gradual programs in AGTLC into programs in the ARL runtime language. Only ARL has a defined evaluation semantics.

To successfully embed the program, elaboration generates evidence objects. These evidence objects carry the information that is needed to support the gradual judgments that type checking relies on. Elaboration uses an initial evidence operation to generate these evidence objects, so the first slicing system I discuss in this chapter handles initial evidence.

5.3.1 *Slicing Initial Evidence*

The initial evidence operation in AGT produces the least precise well-formed evidence object that justifies a particular gradual judgment. AGTLC has only one gradual judgment: consistent equality ($\sim$).

Initial evidence is a partial function, undefined when the inputs are not related by consistent equality. As slicing must be able to accommodate failures, I am interested also in the cases when these partial functions are undefined. Thus instead of partial functions, I am interested in defining total functions that produce a special "failure" output whenever the partial function is undefined. I represent the judgment of the total functions equalling some output as derivation trees encoding both successful and failing computations of the original partial function, so I can apply a uniform slicing process for both.

It may be surprising that I care about failures for initial evidence, because elaboration cares only about successful computations of this operation. Any program where initial evidence fails is not well typed, thus there would be no run-time type errors to deal with, because there would be no program to run. But I still consider slicing for failing initial evidence for two reasons. First, several operations in AGT semantics are treated as partial functions, and run-time errors arise when these functions are undefined. Initial evidence is a simpler context to show the process of dealing with undefined partial functions than other functions involved later in the process of running an AGT gradual program. Second, a key function involved in the evaluation of AGT programs (evidence composition) can sometimes be computed in terms of initial evidence, thus as I discuss in Section 5.6,

some run-time type failures can reduce to failures of initial evidence. Having a slicing system for failing initial evidence may also help in debugging ill-typed programs, as transforming elaboration from a partial function into a total function with explicit failures would amount to generating derivations for ill-typed programs that fail to elaborate or type check. However, designing a slicing system for ill-typed AGT programs is left to future work.

5.3.1.1 *From partial to total initial evidence*

The first step to make initial evidence a total function is to have some representation of failure among evidence objects. AGTLC has a simple definition of evidence objects, which is also discussed in Section 3.1.6. I present evidence objects in a more inductive fashion for slicing as follows:

$$\varepsilon \in \mathrm{Ev}^{\sim} \quad ::= \quad \langle ?, ? \rangle \mid \langle \mathsf{Int}, \mathsf{Int} \rangle \mid \varepsilon \to \varepsilon \quad \text{(evidence objects)}$$

This definition accounts for all well-formed evidence objects in AGTLC.

To obtain total functions from partial functions producing evidence objects, I introduce *total evidence objects*, a representation that explicitly accounts for failure or undefinedness as $\bot$:

$$\varepsilon_{\bot} \in \mathrm{Ev}^{\sim}_{\bot} \quad ::= \quad \varepsilon \mid \bot \quad \text{(total evidence objects)}$$

With total evidence objects, initial evidence can be defined as a total operation. The computation of initial evidence can be represented as derivation trees following the rules in Figure 5.1.

5.3.1.2 *Forward slicing for initial evidence*

A slicing system for initial evidence requires a representation of slices. To provide a syntax for slices I extend the syntax of types and evidences with the $\blacksquare$ marker, which is used to mark parts of a type or an evidence object as irrelevant.

$$\begin{array}{rcll}
\mathring{S} & ::= & \mathsf{Int} \mid ? \mid \mathring{S} \to \mathring{S} \mid \blacksquare & \text{(gradual type slices)} \\
\mathring{\varepsilon} & ::= & \langle ?, ? \rangle \mid \langle \mathsf{Int}, \mathsf{Int} \rangle \mid \mathring{\varepsilon} \to \mathring{\varepsilon} \mid \blacksquare & \text{(evidence slices)} \\
\mathring{\varepsilon}_{\bot} & ::= & \mathring{\varepsilon} \mid \bot & \text{(total evidence slices)}
\end{array}$$

Each gradual type and evidence object has an associated set of slices that may be extracted from it (see the definition of Valid Slices in Section 4.2.2). This set is defined in terms of a partial order for slices, which I index by the original sliced object to avoid confusion with the gradual precision partial order. In AGTLC, this partial order can be presented inductively as follows:

$\boxed{\mathring{S} \sqsubseteq_{S} \mathring{S}}$

$$\frac{}{\blacksquare \sqsubseteq_{S} \mathring{S}} \qquad \frac{}{? \sqsubseteq_{?} ?} \qquad \frac{}{\mathsf{Int} \sqsubseteq_{\mathsf{Int}} \mathsf{Int}} \qquad \frac{\mathring{S}'_1 \sqsubseteq_{S_1} \mathring{S}_1 \qquad \mathring{S}'_2 \sqsubseteq_{S_2} \mathring{S}'_2}{\mathring{S}'_1 \to \mathring{S}'_2 \sqsubseteq_{S_1 \to S_2} \mathring{S}_1 \to \mathring{S}_2}$$

$\boxed{\mathcal{I}[\![S \sim S]\!] : \text{GType} \times \text{GType} \longrightarrow \text{Ev}^{\sim}_{\bot}}$

$$\text{I-Dyn}\ \frac{}{\mathcal{I}[\![? \sim ?]\!] = \langle ?, ? \rangle} \qquad \text{I-Int-B}\ \frac{}{\mathcal{I}[\![\mathsf{Int} \sim \mathsf{Int}]\!] = \langle \mathsf{Int}, \mathsf{Int} \rangle}$$

$$\text{I-Int-L}\ \frac{}{\mathcal{I}[\![\mathsf{Int} \sim ?]\!] = \langle \mathsf{Int}, \mathsf{Int} \rangle} \qquad \text{I-Int-R}\ \frac{}{\mathcal{I}[\![? \sim \mathsf{Int}]\!] = \langle \mathsf{Int}, \mathsf{Int} \rangle}$$

$$\text{I-Fun-Int}\ \frac{}{\mathcal{I}[\![S_1 \to S_2 \sim \mathsf{Int}]\!] = \bot} \qquad \text{I-Int-Fun}\ \frac{}{\mathcal{I}[\![\mathsf{Int} \sim S_1 \to S_2]\!] = \bot}$$

$$\text{I-Fun-B}\ \frac{\mathcal{I}[\![\widetilde{dom}(S_2) \sim \widetilde{dom}(S_1)]\!] = \varepsilon_1 \qquad \mathcal{I}[\![\widetilde{cod}(T_1) \sim \widetilde{cod}(S_2)]\!] = \varepsilon_2}{\mathcal{I}[\![S_1 \sim S_2]\!] = \varepsilon_1 \to \varepsilon_2}$$

$$\text{I-Fun-Fail-D}\ \frac{\mathcal{I}[\![\widetilde{dom}(S_2) \sim \widetilde{dom}(S_1)]\!] = \bot \qquad \mathcal{I}[\![\widetilde{cod}(T_1) \sim \widetilde{cod}(S_2)]\!] = \varepsilon_2}{\mathcal{I}[\![S_1 \sim S_2]\!] = \bot}$$

$$\text{I-Fun-Fail-C}\ \frac{\mathcal{I}[\![\widetilde{dom}(S_2) \sim \widetilde{dom}(S_1)]\!] = \varepsilon_1 \qquad \mathcal{I}[\![\widetilde{cod}(T_1) \sim \widetilde{cod}(S_2)]\!] = \bot}{\mathcal{I}[\![S_1 \sim S_2]\!] = \bot}$$

$$\text{I-Fun-Fail-B}\ \frac{\mathcal{I}[\![\widetilde{dom}(S_2) \sim \widetilde{dom}(S_1)]\!] = \bot \qquad \mathcal{I}[\![\widetilde{cod}(T_1) \sim \widetilde{cod}(S_2)]\!] = \bot}{\mathcal{I}[\![S_1 \sim S_2]\!] = \bot}$$

Figure 5.1: (Total) initial evidence operation for AGTLC.

$\boxed{\varepsilon^{\circ}_{\bot} \sqsubseteq_{\varepsilon_{\bot}} \varepsilon^{\circ}_{\bot}}$

$$\frac{}{\blacksquare \sqsubseteq_{\varepsilon_{\bot}} \varepsilon_{\bot}} \qquad \frac{}{\bot \sqsubseteq_{\bot} \bot} \qquad \frac{}{\langle ?, ? \rangle \sqsubseteq_{\langle ?,? \rangle} \langle ?, ? \rangle} \qquad \frac{}{\langle \mathsf{Int}, \mathsf{Int} \rangle \sqsubseteq_{\langle \mathsf{Int}, \mathsf{Int} \rangle} \langle \mathsf{Int}, \mathsf{Int} \rangle}$$

$$\frac{\varepsilon^{\circ}_{\bot 11} \sqsubseteq_{\varepsilon_{\bot 1}} \varepsilon^{\circ}_{\bot 21} \qquad \varepsilon^{\circ}_{\bot 12} \sqsubseteq_{\varepsilon_{\bot 2}} \varepsilon^{\circ}_{\bot 22}}{\varepsilon^{\circ}_{\bot 11} \to \varepsilon^{\circ}_{\bot 12} \sqsubseteq_{\varepsilon_{\bot 1} \to \varepsilon_{\bot 2}} \varepsilon^{\circ}_{\bot 21} \to \varepsilon^{\circ}_{\bot 22}}$$

I use the partial order to define the domain and range of the slicing functions for initial evidence, following the same calculational approach described in Chapter 4. Because I am able to define forward slicing functions that are well behaved for meets and joins, I rely on the framework of Galois slicing as described in Chapter 4 to obtain a unique backward slicing function, and I focus here in justifying the definition of a forward slicing function for initial evidence.

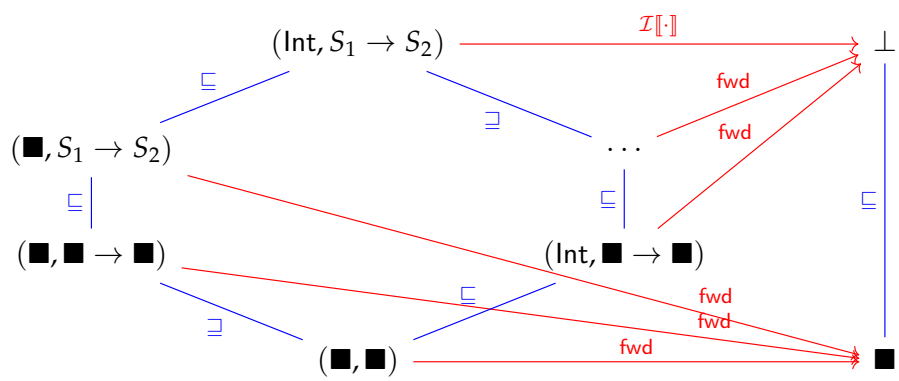
5.3.1.3 *Justifying forward slicing - base cases*

To define forward slicing for any judgment trace, one chooses how much information to preserve on the output given the information that was preserved on the inputs. The specification of forward slicing sets some constraints, but there is still some flexibility on the choices of forward slicing outputs. These choices impact the information that can be recovered later by backward slicing.

When there is flexibility, the main guideline for designing forward slicing is not the formal mathematics of Galois slicing, but whether the slicing functions answer

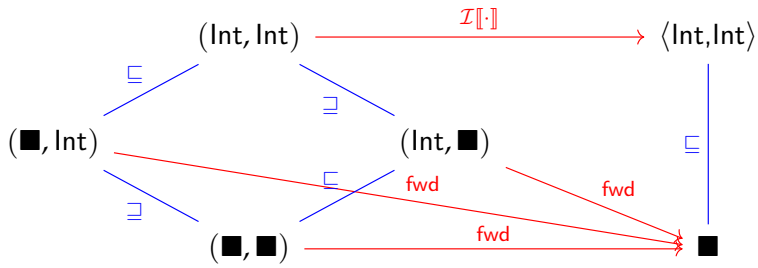
programmers' questions. What is the interpretation of ■ that the designer intends? For the cases that are not directly restricted by the Galois slicing framework, I choose to use the gradual guarantee of Siek, Vitousek, et al. (2015) as a design guideline for assigning meaning to ■ in the context of gradual types and evidence operations. If the dynamic gradual guarantee holds in a gradual language, then run-time type failures in gradual programs can never be fixed by adding more type constraints, only by weakening or removing them. To replicate this guideline, I design forward slicing by interpreting ■ as representing changes in the type constraints. The behaviour of forward slicing then reflects whether changes in the type constraints that are hidden away would affect the structure of the outcome of initial evidence.

Let's begin by applying these guidelines to forward slice traces of the failing evidence composition $\mathcal{I}[\![\mathsf{Int} \sim S_1 \to S_2]\!] = \bot$. In this case, the information that counts is that the head constructors of the two types are incompatible. As when introducing forward slicing in Figure 4.3, I use diagrams encoding the complete partial ordering for slices of the inputs, showing where forward slicing takes them on the partial order of slices of the output:



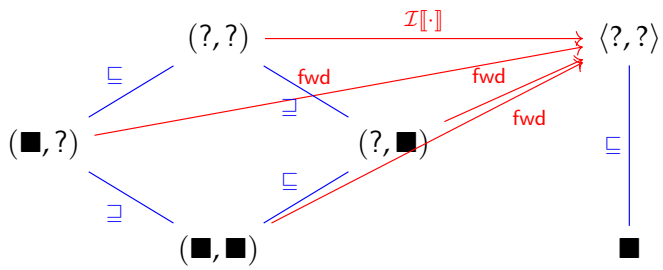
Like in Chapter 4, the top arrow is forced upon us because forward slicing must reproduce the trace when no information is discarded. All other arrows that make forward slicing produce $\bot$ happen when no change in the "irrelevant" parts of the input types could possibly change the output from being a failure. The input has sufficient structural information to always reproduce the failure. Every arrow that makes forward slicing produce ■ is justified by some change being able to prevent a failure. For example, if one used $S_1 \to S_2$ instead of ■ for the inputs $(\blacksquare, S_1 \to S_2)$, then the computation of initial evidence would succeed instead of failing.

A similar argument justifies how forward slicing is defined for slices of the successful computation of initial evidence $\mathcal{I}[\![\mathsf{Int} \sim \mathsf{Int}]\!] = \langle \mathsf{Int}, \mathsf{Int} \rangle$, resulting in the following diagram:



In this diagram, the only way to preserve the successful initial evidence is to keep the complete inputs. In all other cases, some change (say substituting the ■ by some function type) could force initial evidence to fail instead of successfully computing ⟨Int,Int⟩.

The gradual guarantee guideline also impacts how we handle explicit ? gradual types. Because these types represent the weakest possible information, no change to them could affect the result in a way that fixes a failure. Also, initial evidence computations against ? always succeed, so they are never the source of a failure. This approach shows its most extreme face in the slicing of the trace of $\mathcal{I}[\![? \sim ?]\!] = \langle ?, ? \rangle$. Because no change to the types could alter the result, the guidelines always lead forward slicing to $\langle ?, ? \rangle$, no matter whether one slices out an element of the inputs, following this diagram:



This forward slicing definition is compatible with the Galois slicing framework and produces a desirable outcome for backward slicing: computations of initial evidence that result in $\langle ?, ? \rangle$ are not of interest for diagnosing run-time errors, because they cannot be intervened on in any way, so the inputs are always marked as irrelevant (■).

Gradual typing experts may observe a connection between the way I treat the unknown type ? when slicing initial evidence (which is used in slicing evidence composition) and the blame calculus approach that, at run time, discards cast labels whenever reducing casts that lose precision. To observe the differences, consider the following reduction rules from Wadler and Findler (2009) (Fig. 4):

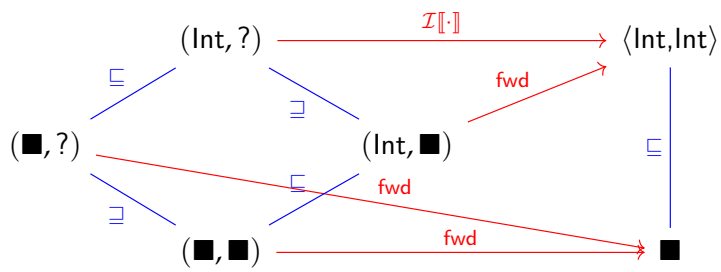
$$E[\langle \mathtt{Dyn} \Leftarrow \mathtt{Dyn} \rangle^{p}\ v] \longrightarrow E[v] \tag{5}$$

$$E[\langle \mathtt{Dyn} \Leftarrow B \rangle^{p}\ v] \longrightarrow E[\mathtt{Dyn}_B(v)] \tag{6}$$

$$E[\langle T \Leftarrow \mathtt{Dyn} \rangle^{p}\ \mathtt{Dyn}_G(v)] \longrightarrow \mathtt{blame}\ p \quad \text{if } T \not\sim G \tag{9}$$

Wadler and Findler (2009) uses program location markers p to reference some part of a program in an error message. These locations are connected to casts and must be propagated as programs evaluate. Rule (5) drops the highlighted label of a $\langle \mathtt{Dyn} \Leftarrow \mathtt{Dyn} \rangle$ cast, which implies that it cannot be part of a later error message. This approach has a similar effect to slicing for $\mathcal{I}[\![? \sim ?]\!] = \langle ?, ? \rangle$, where we end up always marking the inputs as irrelevant. Although I only discuss evidence composition in Section 5.6.2, here I'd like to mention that this initial evidence design transfers to evidence composition against a $\langle ?, ? \rangle$ evidence object. When $\langle ?, ? \rangle$ is an input to an evidence composition, backward slicing will always mark that input as irrelevant.

The key differences between the runtime semantics of errors in the blame calculus and program slicing can be compared by focusing on the difference between rules (6), (9), and the following slicing diagram for $\mathcal{I}[\![\mathsf{Int} \sim ?]\!] = \langle \mathsf{Int}, \mathsf{Int} \rangle$:



Consider the following program extract, which intends to represent a longer program. To connect with the blame calculus semantics, I give the type ascription appearing in each line a location identifier, which I place to the left of the program.

$$
\begin{array}{ll}
(p_1) & \mathsf{let}\ (x :?) = (12 :: ?)\ \mathsf{in} \\
 & \ldots \\
(p_2) & \mathsf{let}\ (f :?) = (x :: ?)\ \mathsf{in} \\
 & \quad \ldots \\
(p_3) & \quad (f :: \mathsf{Int} \to \mathsf{Int})\ 5
\end{array}
$$

When we run this program with the blame calculus semantics, rules (5) and (6) drop the information about p_1 and p_2: the labels (highlighted) appear on the input side of the rule but not on the output side. The blame calculus assumes that losing precision is a safe operation that never fails, so the error message reports only p_3. The programmer is not informed about p_1, which introduced the other half of the inconsistency triggering the failure.

Unlike the error report in a blame semantics, slicing preserves the parts of a program where types get weakened but that impose constraints relevant to a run-time type error. With program slicing, the only information that is discarded in this example is related to p_2.

My approach to slicing marks as irrelevant only the parts of types that impose no constraints. As any change to the ascription going from ? to ? at p_2 cannot make the error go away, it can be safely marked as irrelevant. But since the type ascription at p_1

$$\boxed{\underset{\mathcal{I}[\![S_1 \sim S_2]\!]=\varepsilon_\bot}{\mathsf{fwd}} : (\textsc{slices}[S_1] \times \textsc{slices}[S_2]) \longrightarrow \textsc{slices}[\varepsilon_\bot]}$$

$$\frac{}{\underset{\mathcal{I}[\![?\sim ?]\!]=\langle ?,?\rangle}{\mathsf{fwd}} \left(\mathring{S}_1, \mathring{S}_2\right) = \langle ?, ?\rangle}$$

$$\frac{\mathring{S}_1 \sqcap \mathring{S}_2 = \mathsf{Int}}{\underset{\mathcal{I}[\![\mathsf{Int}\sim\mathsf{Int}]\!]=\langle \mathsf{Int},\mathsf{Int}\rangle}{\mathsf{fwd}} \left(\mathring{S}_1, \mathring{S}_2\right) = \langle \mathsf{Int},\mathsf{Int}\rangle} \qquad \frac{\mathring{S}_1 \sqcap \mathring{S}_2 = \blacksquare}{\underset{\mathcal{I}[\![\mathsf{Int}\sim\mathsf{Int}]\!]=\langle \mathsf{Int},\mathsf{Int}\rangle}{\mathsf{fwd}} \left(\mathring{S}_1, \mathring{S}_2\right) = \blacksquare}$$

$$\frac{}{\underset{\mathcal{I}[\![\mathsf{Int}\sim ?]\!]=\langle \mathsf{Int},\mathsf{Int}\rangle}{\mathsf{fwd}} \left(\mathsf{Int}, \mathring{S}_2\right) = \langle \mathsf{Int},\mathsf{Int}\rangle} \qquad \frac{}{\underset{\mathcal{I}[\![\mathsf{Int}\sim ?]\!]=\langle \mathsf{Int},\mathsf{Int}\rangle}{\mathsf{fwd}} \left(\blacksquare, \mathring{S}_2\right) = \blacksquare}$$

$$\frac{}{\underset{\mathcal{I}[\![?\sim \mathsf{Int}]\!]=\langle \mathsf{Int},\mathsf{Int}\rangle}{\mathsf{fwd}} \left(\mathring{S}_1, \mathsf{Int}\right) = \langle \mathsf{Int},\mathsf{Int}\rangle} \qquad \frac{}{\underset{\mathcal{I}[\![?\sim \mathsf{Int}]\!]=\langle \mathsf{Int},\mathsf{Int}\rangle}{\mathsf{fwd}} \left(\mathring{S}_1, \blacksquare\right) = \blacksquare}$$

$$\frac{}{\underset{\mathcal{I}[\![\mathsf{Int}\sim S_1\to S_2]\!]=\bot}{\mathsf{fwd}} \left(\mathsf{Int}, \mathring{S}_1 \to \mathring{S}_2\right) = \bot}$$

$$\frac{}{\underset{\mathcal{I}[\![\mathsf{Int}\sim S_1\to S_2]\!]=\bot}{\mathsf{fwd}} \left(\blacksquare, \mathring{S}_1 \to \mathring{S}_2\right) = \blacksquare} \qquad \frac{}{\underset{\mathcal{I}[\![\mathsf{Int}\sim S_1\to S_2]\!]=\bot}{\mathsf{fwd}} (\mathsf{Int}, \blacksquare) = \blacksquare}$$

$$\frac{}{\underset{\mathcal{I}[\![S_1\to S_2\sim \mathsf{Int}]\!]=\bot}{\mathsf{fwd}} \left(\mathring{S}_1 \to \mathring{S}_2, \mathsf{Int}\right) = \bot} \qquad \frac{}{\underset{\mathcal{I}[\![S_1\to S_2\sim \mathsf{Int}]\!]=\bot}{\mathsf{fwd}} (\blacksquare, \mathsf{Int}) = \blacksquare}$$

$$\frac{}{\underset{\mathcal{I}[\![S_1\to S_2\sim \mathsf{Int}]\!]=\bot}{\mathsf{fwd}} \left(\mathring{S}_1 \to \mathring{S}_2, \blacksquare\right) = \blacksquare}$$

Figure 5.2: Forward slicing for (total) initial evidence in AGTLC, part 1

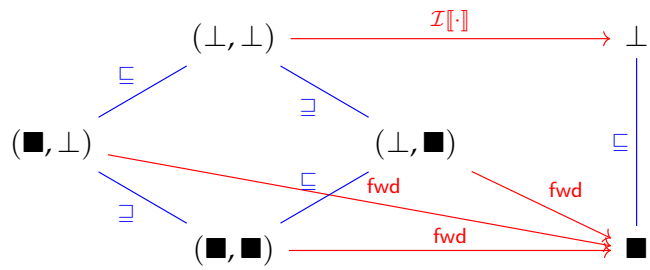
introduces a constraint required to produce the failure, backward slicing preserves it. Slicing also indicates that the relevant slice for the type at p_3 is $\blacksquare \to \blacksquare$, so that ascribing f to any function type would still produce the failure.

These guidelines suffice to specify forward slicing for initial evidence of all traces that have no premises (those that, when seen as derivation trees, are leaves). I present the rules for forward slicing these cases in Figure 5.2.

5.3.1.4 *Justifying forward slicing: inductive rules*

After forward slicing for the base cases of initial evidence, I can discuss slicing for the initial evidence rules that include premises. Rule [I-Fun-B] in Figure 5.1 represents a successful computation and inductively depends on slicing each premise. There are three initial evidence rules in Figure 5.1 that deal with failures and have premises in them. Rules [I-Fun-Fail-D] and [I-Fun-Fail-C] have only one failing premise, which suffices to reproduce the failure. These rules preserve information similarly to what a "multiplication by zero" rule would preserve: the other premise does not affect the outcome.

The last rule is [I-Fun-Fail-B], which has two failing premises. How do we slice the cases when one of the failures is marked as irrelevant? Consider an initial evidence computation trace for $\mathcal{I}[\![S_1 \to S_2 \sim S_3 \to S_4]\!] = \bot$ when both $\mathcal{I}[\![S_3 \sim S_1]\!] = \bot$ and $\mathcal{I}[\![S_2 \sim S_4]\!] = \bot$. To focus the reader's attention on the multiple failures in premises, I diagram each of the premises as resulting in $\bot$, much as if computing $\mathcal{I}[\![\bot \sim \bot]\!]$, which is outside the domain of initial evidence.



The key reason to choose this forward slicing approach is that it preserves both failures. The only other option would be to forward slice to a failure when some of the premises are marked as irrelevant, but then slicing would prioritize one failure over another. By discarding some information, slicing would break the over approximation guideline for failures.

Choosing to disregard part of the failure justification would also impact backward slicing, which would not preserve all the inconsistencies in the original program. Choosing some failures over others would give programmers incomplete information about the inconsistencies leading to the failure. If we allow one of the premises to be sufficient to reproduce the failure (that is, $(\blacksquare, \bot)$ or $(\bot, \blacksquare)$) by allowing it to forward slice to $\bot$ instead of $\blacksquare$, then backward slicing for $\bot$ would be forced to mark the other failure premise as irrelevant, instead of keeping both $(\bot, \bot)$. This approach would discard possibly relevant information and direct the developer to a particular part of the failure, an approach I argued against in Section 2.5.3.

This design guideline justifies the rules in Figure 5.3, describing forward slicing for initial evidence computations that have some structure. Given the simplicity of the types in AGTLC, the only cases that have structure are those where at least one of the gradual types involved is a function. To make the rules more succinct, I overload the functions $\widetilde{dom}$ and $\widetilde{cod}$ to accommodate slices: they remain partial functions as previously defined, but they are also defined for $\blacksquare$, where $\widetilde{dom}(\blacksquare) = \blacksquare$ and $\widetilde{cod}(\blacksquare) = \blacksquare$.

Rules [FI-Fun-Succ-Keep] and [FI-Fun-Succ-Drop] in Figure 5.3 address the cases where both premises successfully compute an initial evidence. When no premise provides useful information (they both produce $\blacksquare$), forward slicing produces $\blacksquare$ ([FI-Fun-Succ-Drop]). Otherwise, I propagate the information obtained recursively via forward slicing and the structure of the evidence object ([FI-Fun-Succ-Keep]). Rules [FI-Fun-Fail-L] and [FI-Fun-Fail-R] address cases with a single failure on the premises, disregarding the successful premise which is not involved in the failure, and propagating the information from the failing premise. Finally, rule [FI-Fun-Fail-B] enforces that whenever two simultaneous

$$\text{FI-Fun-Succ-Keep}\ \frac{\begin{array}{c}\underset{\mathcal{I}[\![\widetilde{dom}(S_2)\sim\widetilde{dom}(S_1)]\!]=\varepsilon_3}{\mathsf{fwd}}\left(\widetilde{dom}(\mathring{S}_2),\widetilde{dom}(\mathring{S}_1)\right)=\mathring{\varepsilon}_3\\ \underset{\mathcal{I}[\![\widetilde{cod}(S_1)\sim\widetilde{cod}(S_2)]\!]=\varepsilon_4}{\mathsf{fwd}}\left(\widetilde{cod}(\mathring{S}_1),\widetilde{cod}(\mathring{S}_2)\right)=\mathring{\varepsilon}_4\\ \mathring{\varepsilon}_3\neq\blacksquare\text{ or }\mathring{\varepsilon}_4\neq\blacksquare \qquad S_1\neq ?\text{ or }S_2\neq ?\end{array}}{\underset{\mathcal{I}[\![S_1\sim S_2]\!]=\varepsilon_3\to\varepsilon_4}{\mathsf{fwd}}\left(\mathring{S}_1,\mathring{S}_2\right)=\mathring{\varepsilon}_3\to\mathring{\varepsilon}_4}$$

$$\text{FI-Fun-Succ-Drop}\ \frac{\begin{array}{c}\underset{\mathcal{I}[\![\widetilde{dom}(S_2)\sim\widetilde{dom}(S_1)]\!]=\varepsilon_3}{\mathsf{fwd}}\left(\widetilde{dom}(\mathring{S}_2),\widetilde{dom}(\mathring{S}_1)\right)=\blacksquare\\ \underset{\mathcal{I}[\![\widetilde{cod}(S_1)\sim\widetilde{cod}(S_2)]\!]=\varepsilon_4}{\mathsf{fwd}}\left(\widetilde{cod}(\mathring{S}_1),\widetilde{cod}(\mathring{S}_2)\right)=\blacksquare \qquad S_1\neq ?\text{ or }S_2\neq ?\end{array}}{\underset{\mathcal{I}[\![S_1\sim S_2]\!]=\varepsilon_3\to\varepsilon_4}{\mathsf{fwd}}\left(\mathring{S}_1,\mathring{S}_2\right)=\blacksquare}$$

$$\text{FI-Fun-Fail-L}\ \frac{\begin{array}{c}\underset{\mathcal{I}[\![\widetilde{dom}(S_2)\sim\widetilde{dom}(S_1)]\!]=\bot}{\mathsf{fwd}}\left(\widetilde{dom}(\mathring{S}_2),\widetilde{dom}(\mathring{S}_1)\right)=\varepsilon^{\circ}_{\bot}\\ \underset{\mathcal{I}[\![\widetilde{cod}(S_1)\sim\widetilde{cod}(S_2)]\!]=\varepsilon_4}{\mathsf{fwd}}\left(\widetilde{cod}(\mathring{S}_1),\widetilde{cod}(\mathring{S}_2)\right)=\mathring{\varepsilon}_4 \qquad S_1\neq ?\text{ or }S_2\neq ?\end{array}}{\underset{\mathcal{I}[\![S_1\sim S_2]\!]=\bot}{\mathsf{fwd}}\left(\mathring{S}_1,\mathring{S}_2\right)=\varepsilon^{\circ}_{\bot}}$$

$$\text{FI-Fun-Fail-R}\ \frac{\begin{array}{c}\underset{\mathcal{I}[\![\widetilde{dom}(S_2)\sim\widetilde{dom}(S_1)]\!]=\varepsilon_3}{\mathsf{fwd}}\left(\widetilde{dom}(\mathring{S}_2),\widetilde{dom}(\mathring{S}_1)\right)=\mathring{\varepsilon}_3\\ \underset{\mathcal{I}[\![\widetilde{cod}(S_1)\sim\widetilde{cod}(S_2)]\!]=\bot}{\mathsf{fwd}}\left(\widetilde{cod}(\mathring{S}_1),\widetilde{cod}(\mathring{S}_2)\right)=\varepsilon^{\circ}_{\bot} \qquad S_1\neq ?\text{ or }S_2\neq ?\end{array}}{\underset{\mathcal{I}[\![S_1\sim S_2]\!]=\varepsilon_{\bot}}{\mathsf{fwd}}\left(\mathring{S}_1,\mathring{S}_2\right)=\varepsilon^{\circ}_{\bot}}$$

$$\text{FI-Fun-Fail-B}\ \frac{\begin{array}{c}\underset{\mathcal{I}[\![\widetilde{dom}(S_2)\sim\widetilde{dom}(S_1)]\!]=\bot}{\mathsf{fwd}}\left(\widetilde{dom}(\mathring{S}_2),\widetilde{dom}(\mathring{S}_1)\right)=\varepsilon^{\circ}_{\bot 1}\\ \underset{\mathcal{I}[\![\widetilde{cod}(S_1)\sim\widetilde{cod}(S_2)]\!]=\bot}{\mathsf{fwd}}\left(\widetilde{cod}(\mathring{S}_1),\widetilde{cod}(\mathring{S}_2)\right)=\varepsilon^{\circ}_{\bot 2} \qquad S_1\neq ?\text{ or }S_2\neq ?\end{array}}{\underset{\mathcal{I}[\![S_1\sim S_2]\!]=\varepsilon_{\bot}}{\mathsf{fwd}}\left(\mathring{S}_1,\mathring{S}_2\right)=\varepsilon^{\circ}_{\bot 1}\sqcap\varepsilon^{\circ}_{\bot 2}}$$

Figure 5.3: Forward Slicing for (Total) Initial evidence in AGTLC, part 2

failures of initial evidence happen in the premises, I only forward slice the slice to $\bot$ if both premises also forward slice their slices to $\bot$.

This definition of forward slicing for initial evidence satisfies all the requirements previously established in Definition 16 and Theorem 26 of Chapter 4:

Theorem 35 (Initial evidence forward slicing is well behaved for meets and joins)**.** *For any trace* $\overset{\mathcal{D}}{\mathcal{I}[\![S_1 \sim S_2]\!] = \varepsilon_\bot}$ *the function* $\underset{\mathcal{I}[\![S_1 \sim S_2]\!] = \varepsilon_\bot}{\mathsf{fwd}} \left(\mathring{S}_1, \mathring{S}_2\right) = \mathring{\varepsilon}_\bot$ *is a forward slicing function well behaved for meets and joins. Thus it uniquely determines its backward slicing function.*

5.3.2 *Slicing elaboration*

The slicing system for initial evidence can be used to define slicing for the elaboration algorithm of AGTLC, which is presented in Section 3.1.8. Slicing elaboration is necessary to connect the programs the programmer writes in the source language with their run-time representation and failures arising in runtime terms with the programs the programmer writes in the source language. Without a slicing system for elaboration, users would need to be intimately familiar with the intermediate representation of programs and connect the parts of the intermediate representation to the original program themselves.

As in any slicing system, the slicing functions for elaboration traces distinguish between what is input and output in the elaboration procedure: for forward slicing, the context and source term slices are inputs, while the runtime term and the type are outputs. For backward slicing, outputs and inputs are reversed: the runtime term and type are inputs, and the outputs are a slice of the context and a slice of the source term. These constraints are equivalent to requiring the slicing functions for elaboration to have the following signatures:

$$\underset{\Gamma \vdash t \rightsquigarrow \mathbf{e} : S}{\mathsf{fwd}} : \textsc{slices}[\Gamma] \times \textsc{slices}[t] \longrightarrow \textsc{slices}[\mathbf{e}] \times \textsc{slices}[S]$$

$$\underset{\Gamma \vdash t \rightsquigarrow \mathbf{e} : S}{\mathsf{bwd}} : \textsc{slices}[\mathbf{e}] \times \textsc{slices}[S] \longrightarrow \textsc{slices}[\Gamma] \times \textsc{slices}[t]$$

I present the forward slicing algorithm in Figure 5.4. The rules follow the structure of the trace, and I have placed together all the rules that deal with ■ slices of terms.

Theorem 36 (Elaboration forward slicing is well behaved for meets and joins)**.** *For any trace* $\overset{\mathcal{D}}{\Gamma \vdash t \rightsquigarrow \mathbf{e} : S}$ *the function* $\underset{\Gamma \vdash t \rightsquigarrow \mathbf{e} : S}{\mathsf{fwd}} \left(\mathring{\Gamma}, \mathring{t}\right) = \left(\mathring{\mathbf{e}}, \mathring{S}\right)$ *is a forward slicing function well behaved for meets and joins. Thus it uniquely determines its backward slicing function.*

5.4 SLICING EVIDENCE COMPOSITION FOR AGTLC

A slicing system for the runtime language that works in the presence of run-time type failures must also address the mechanisms that propagate run-time type information. I propagate successful run-time type information to ensure that the information that triggers a failure is present when forward slicing gets to the point of triggering the

$$(\rightsquigarrow\text{x})\ \frac{x : \mathring{S} \in \mathring{\Gamma}}{\underset{\Gamma \vdash x \rightsquigarrow \mathbf{x} : S}{\mathsf{fwd}} \left(\mathring{\Gamma}, x\right) = \left(\mathbf{x}, \mathring{S}\right)} \qquad (\rightsquigarrow\text{n})\ \frac{}{\underset{\Gamma \vdash n \rightsquigarrow \mathbf{n} : \mathsf{Int}}{\mathsf{fwd}} \left(\mathring{\Gamma}, n\right) = \left(\mathbf{n}, \mathsf{Int}\right)}$$

$$(\rightsquigarrow\text{app})\ \frac{\begin{array}{c} \underset{\Gamma \vdash t_1 \rightsquigarrow \mathbf{e}_1 : S_1}{\mathsf{fwd}} \left(\mathring{\Gamma}, \mathring{t}_1\right) = \left(\mathring{\mathbf{e}}_1, \mathring{S}_1\right) \quad \underset{\Gamma \vdash t_2 \rightsquigarrow \mathbf{e}_2 : S_2}{\mathsf{fwd}} \left(\mathring{\Gamma}, \mathring{t}_2\right) = \left(\mathring{\mathbf{e}}_2, \mathring{S}_2\right) \\ \underset{\mathcal{I}[\![S_1 \sim \widetilde{dom}(S_1) \to \widetilde{cod}(S_1)]\!] = \varepsilon_1}{\mathsf{fwd}} \left(\mathring{S}_1, \widetilde{dom}(\mathring{S}_1) \to \widetilde{cod}(\mathring{S}_1)\right) = \mathring{\varepsilon}_1 \\ \underset{\mathcal{I}[\![S_2 \sim \widetilde{dom}(S_1)]\!] = \varepsilon_2}{\mathsf{fwd}} \left(\mathring{S}_2, \widetilde{dom}(\mathring{S}_1)\right) = \mathring{\varepsilon}_2 \end{array}}{\underset{\Gamma \vdash t_1\ t_2 \rightsquigarrow \varepsilon_1 \mathbf{e}_1\ \varepsilon_2 \mathbf{e}_2 : \widetilde{cod}(S_1)}{\mathsf{fwd}} \left(\mathring{\Gamma}, \mathring{t}_1\ \mathring{t}_2\right) = \left(\mathring{\varepsilon}_1 \mathring{\mathbf{e}}_1\ \mathring{\varepsilon}_2 \mathring{\mathbf{e}}_2, \widetilde{cod}(\mathring{S}_1)\right)}$$

$$(\rightsquigarrow\lambda)\ \frac{\underset{\Gamma, x : S_1 \vdash t \rightsquigarrow \mathbf{e} : S_2}{\mathsf{fwd}} \left(\mathring{\Gamma}, x : \mathring{S}_1, \mathring{t}\right) = \left(\mathring{\mathbf{e}}, \mathring{S}_2\right)}{\underset{\Gamma \vdash (\lambda x : S_1 . t) \rightsquigarrow (\lambda \mathbf{x} . \mathbf{e}) : S_1 \to S_2}{\mathsf{fwd}} \left(\mathring{\Gamma}, (\lambda x : \mathring{S}_1 . \mathring{t})\right) = \left(\lambda \mathbf{x} . \mathring{\mathbf{e}}, \mathring{S}_1 \to \mathring{S}_2\right)}$$

$$(\rightsquigarrow::)\ \frac{\underset{\Gamma \vdash t \rightsquigarrow \mathbf{e} : S}{\mathsf{fwd}} \left(\mathring{\Gamma}, \mathring{t}\right) = \left(\mathring{\mathbf{e}}, \mathring{S}\right) \quad \underset{\mathcal{I}[\![S \sim S_1]\!] = \varepsilon}{\mathsf{fwd}} \left(\mathring{S}, \mathring{S}_1\right) = \mathring{\varepsilon}}{\underset{\Gamma \vdash t :: S_1 \rightsquigarrow \varepsilon \mathbf{e} : S_1}{\mathsf{fwd}} \left(\mathring{\Gamma}, \mathring{t} :: \mathring{S}_1\right) = \left(\mathring{\varepsilon} \mathring{\mathbf{e}}, \mathring{S}_1\right)}$$

$$(\rightsquigarrow\text{x-}\blacksquare)\ \frac{}{\underset{\Gamma \vdash x \rightsquigarrow \mathbf{x} : S}{\mathsf{fwd}} \left(\mathring{\Gamma}, \blacksquare\right) = (\blacksquare, \blacksquare)} \qquad (\rightsquigarrow\text{n-}\blacksquare)\ \frac{}{\underset{\Gamma \vdash n \rightsquigarrow \mathbf{n} : \mathsf{Int}}{\mathsf{fwd}} \left(\mathring{\Gamma}, \blacksquare\right) = (\blacksquare, \blacksquare)}$$

$$(\rightsquigarrow\text{app-}\blacksquare)\ \frac{}{\underset{\Gamma \vdash t_1\ t_2 \rightsquigarrow \varepsilon_1 \mathbf{e}_1\ \varepsilon_2 \mathbf{e}_2 : \widetilde{cod}(S_1)}{\mathsf{fwd}} \left(\mathring{\Gamma}, \blacksquare\right) = (\blacksquare, \blacksquare)}$$

$$(\rightsquigarrow\lambda\text{-}\blacksquare)\ \frac{}{\underset{\Gamma \vdash (\lambda x : S_1 . t) \rightsquigarrow (\lambda \mathbf{x} . \mathbf{e}) : S_1 \to S_2}{\mathsf{fwd}} \left(\mathring{\Gamma}, \blacksquare\right) = (\blacksquare, \blacksquare)} \qquad (\rightsquigarrow::\text{-}\blacksquare)\ \frac{}{\underset{\Gamma \vdash t :: S_1 \rightsquigarrow \varepsilon \mathbf{e} : S_1}{\mathsf{fwd}} \left(\mathring{\Gamma}, \blacksquare\right) = (\blacksquare, \blacksquare)}$$

Figure 5.4: Forward slicing for Elaboration for AGTLC

$\boxed{\varepsilon;\varepsilon = \varepsilon_\bot}$

$$\text{Dyn-Dyn}\ \frac{}{\langle ?,?\rangle;\langle ?,?\rangle = \langle ?,?\rangle} \qquad \text{Dyn-Int}\ \frac{}{\langle ?,?\rangle;\langle \mathsf{Int},\mathsf{Int}\rangle = \langle \mathsf{Int},\mathsf{Int}\rangle}$$

$$\text{Int-Dyn}\ \frac{}{\langle \mathsf{Int},\mathsf{Int}\rangle;\langle ?,?\rangle = \langle \mathsf{Int},\mathsf{Int}\rangle} \qquad \text{Int-Int}\ \frac{}{\langle \mathsf{Int},\mathsf{Int}\rangle;\langle \mathsf{Int},\mathsf{Int}\rangle = \langle \mathsf{Int},\mathsf{Int}\rangle}$$

$$\text{Int-Fun}\ \frac{}{\langle \mathsf{Int},\mathsf{Int}\rangle;\varepsilon_3 \to \varepsilon_4 = \bot} \qquad \text{Fun-Int}\ \frac{}{\varepsilon_1 \to \varepsilon_2;\langle \mathsf{Int},\mathsf{Int}\rangle = \bot}$$

$$\text{Fun-Pass}\ \frac{\begin{array}{c}\mathrm{idom}(\varepsilon_2);\mathrm{idom}(\varepsilon_1) = \varepsilon_5 \quad \mathrm{icod}(\varepsilon_1);\mathrm{icod}(\varepsilon_2) = \varepsilon_6 \\ \varepsilon_1 \neq \langle ?,?\rangle \text{ or } \varepsilon_2 \neq \langle ?,?\rangle\end{array}}{\varepsilon_1;\varepsilon_2 = \varepsilon_5 \to \varepsilon_6}$$

$$\text{Fun-Err-l}\ \frac{\begin{array}{c}\mathrm{idom}(\varepsilon_2);\mathrm{idom}(\varepsilon_1) = \bot \quad \mathrm{icod}(\varepsilon_1);\mathrm{icod}(\varepsilon_2) = \varepsilon_6 \\ \varepsilon_1 \neq \langle ?,?\rangle \text{ or } \varepsilon_2 \neq \langle ?,?\rangle\end{array}}{\varepsilon_1;\varepsilon_2 = \bot}$$

$$\text{Fun-Err-r}\ \frac{\begin{array}{c}\mathrm{idom}(\varepsilon_2);\mathrm{idom}(\varepsilon_1) = \varepsilon_5 \quad \mathrm{icod}(\varepsilon_1);\mathrm{icod}(\varepsilon_2) = \bot \\ \varepsilon_1 \neq \langle ?,?\rangle \text{ or } \varepsilon_2 \neq \langle ?,?\rangle\end{array}}{\varepsilon_1;\varepsilon_2 = \bot}$$

$$\text{Fun-Err-both}\ \frac{\begin{array}{c}\mathrm{idom}(\varepsilon_2);\mathrm{idom}(\varepsilon_1) = \bot \quad \mathrm{icod}(\varepsilon_1);\mathrm{icod}(\varepsilon_2) = \bot \\ \varepsilon_1 \neq \langle ?,?\rangle \text{ or } \varepsilon_2 \neq \langle ?,?\rangle\end{array}}{\varepsilon_1;\varepsilon_2 = \bot}$$

Figure 5.5: (Total) Evidence Composition for AGTLC

failure. This information is propagated using evidence composition, so slicing evidence composition is a key step in providing a slicing system that presents the inconsistencies that triggered some run-time type check failure.

Like when processing initial evidence, first I present evidence composition as a total function with derivation trees that represent their computation traces. Figure 5.5 describes the computation of evidence composition for AGTLC as an inductive total judgment. These derivation rules represent every possible trace of the computation of evidence composition. Note that this definition has some properties of interest, similar to those in the presentation of initial evidence:

- It is a *total* function.
- There's a unique derivation for each possible computation of composition.
- It deals separately with errors and non-errors: the rules never attempt to match against $\varepsilon_\bot$! Whenever a choice must be made between a valid evidence ε and an error $\bot$, there are separate rules to deal with each.
- It relies on operations idom and icod to present uniform rules that deal with functions, including the case when one of the composed evidence objects contains no function types but is $\langle ?,?\rangle$ instead. The premises of function-related rules always check that at least one of the evidence objects is not $\langle ?,?\rangle$, so that the evidence

$$\boxed{\underset{\varepsilon_1;\varepsilon_2=\varepsilon_\perp}{\mathsf{fwd}} : (\textsc{slices}[\varepsilon_1] \times \textsc{slices}[\varepsilon_2]) \longrightarrow \textsc{slices}[\varepsilon_\perp]}$$

$$\text{Dyn-Dyn}\ \frac{}{\underset{\langle ?,?\rangle;\langle ?,?\rangle=\langle ?,?\rangle}{\mathsf{fwd}}\ (\mathring{\varepsilon}_1, \mathring{\varepsilon}_2) = \langle ?, ?\rangle}$$

$$\text{Dyn-Int}\ \frac{}{\underset{\langle ?,?\rangle;\langle \mathsf{Int},\mathsf{Int}\rangle=\langle \mathsf{Int},\mathsf{Int}\rangle}{\mathsf{fwd}}\ (\mathring{\varepsilon}_1, \mathring{\varepsilon}_2) = \mathring{\varepsilon}_2} \qquad \text{Int-Dyn}\ \frac{}{\underset{\langle \mathsf{Int},\mathsf{Int}\rangle;\langle ?,?\rangle=\langle \mathsf{Int},\mathsf{Int}\rangle}{\mathsf{fwd}}\ (\mathring{\varepsilon}_1, \mathring{\varepsilon}_2) = \mathring{\varepsilon}_1}$$

$$\text{Int-Int}\ \frac{}{\underset{\langle \mathsf{Int},\mathsf{Int}\rangle;\langle \mathsf{Int},\mathsf{Int}\rangle=\langle \mathsf{Int},\mathsf{Int}\rangle}{\mathsf{fwd}}\ (\varepsilon_{\perp 1}, \varepsilon_{\perp 2}) = \varepsilon_{\perp 1} \sqcap \varepsilon_{\perp 2}}$$

$$\text{Int-Fun-Missing}\ \frac{\mathring{\varepsilon}_1 = \blacksquare \text{ or } \mathring{\varepsilon}_2 = \blacksquare}{\underset{\langle \mathsf{Int},\mathsf{Int}\rangle;\varepsilon_1\to\varepsilon_2=\perp}{\mathsf{fwd}}\ (\mathring{\varepsilon}_1, \mathring{\varepsilon}_2) = \blacksquare}$$

$$\text{Int-Fun-Enough}\ \frac{}{\underset{\langle \mathsf{Int},\mathsf{Int}\rangle;\varepsilon_1\to\varepsilon_2=\perp}{\mathsf{fwd}}\ (\langle \mathsf{Int},\mathsf{Int}\rangle, \mathring{\varepsilon}_1 \to \mathring{\varepsilon}_2) = \perp}$$

$$\text{Fun-Int-Missing}\ \frac{\mathring{\varepsilon}_1 = \blacksquare \text{ or } \mathring{\varepsilon}_2 = \blacksquare}{\underset{\varepsilon_1\to\varepsilon_2;\langle \mathsf{Int},\mathsf{Int}\rangle=\perp}{\mathsf{fwd}}\ (\mathring{\varepsilon}_1, \mathring{\varepsilon}_2) = \blacksquare}$$

$$\text{Fun-Int-Enough}\ \frac{}{\underset{\varepsilon_1\to\varepsilon_2;\langle \mathsf{Int},\mathsf{Int}\rangle=\perp}{\mathsf{fwd}}\ (\mathring{\varepsilon}_1 \to \mathring{\varepsilon}_2, \langle \mathsf{Int},\mathsf{Int}\rangle) = \perp}$$

Figure 5.6: Forward Slicing for Evidence Composition in AGTLC, part 1

composition trace for the computation of $\langle ?, ?\rangle; \langle ?, ?\rangle$ remains unique, using the [Dyn-Dyn] rule.

I use the definition of evidence composition in Figure 5.5 as the starting point to present forward slicing for evidence composition.

5.4.1 *Forward slicing for evidence composition*

One can follow the same guidelines used to provide forward slicing for initial evidence to provide forward slicing for evidence composition. Those guidelines justify the definition of forward slicing in Figures 5.6 and 5.7. Evidence composition follows a similar structure to the definition of initial evidence, while using evidence objects instead of gradual types as inputs. All the rules in Figure 5.6 mirror the "base rules" described in Figure 5.2 and Section 5.3.1.2. The rules in Figure 5.7 mirror the "inductive rules" discussed in Figure 5.3. When evidence composition succeeds, rules preserve as much structure as available in the input slices. When evidence composition fails, rules preserve sufficient information to reproduce the failure, ensuring that whenever multiple failures are detected simultaneously in premises, all the failures are included.

This definition of forward slicing for evidence composition satisfies all the requirements previously established in Definition 16 and Theorem 26 of Chapter 4:

$$\text{Fun-Err-both}\ \frac{\begin{array}{c}\varepsilon_1 \neq \langle ?, ? \rangle \text{ or } \varepsilon_2 \neq \langle ?, ? \rangle \\ \mathsf{fwd}_{\mathrm{idom}(\varepsilon_2);\mathrm{idom}(\varepsilon_1)=\bot}\ (\mathrm{idom}(\mathring{\varepsilon}_2), \mathrm{idom}(\mathring{\varepsilon}_1)) = \mathring{\varepsilon}_{\bot 5} \\ \mathsf{fwd}_{\mathrm{icod}(\varepsilon_2);\mathrm{icod}(\varepsilon_2)=\bot}\ (\mathrm{icod}(\mathring{\varepsilon}_1), \mathrm{icod}(\mathring{\varepsilon}_2)) = \mathring{\varepsilon}_{\bot 6}\end{array}}{\mathsf{fwd}_{\varepsilon_1;\varepsilon_2=\bot}\ (\mathring{\varepsilon}_1, \mathring{\varepsilon}_2) = \mathring{\varepsilon}_{\bot 5} \sqcap \mathring{\varepsilon}_{\bot 6}}$$

$$\text{Fun-Err-l}\ \frac{\begin{array}{c}\varepsilon_1 \neq \langle ?, ? \rangle \text{ or } \varepsilon_2 \neq \langle ?, ? \rangle \\ \mathsf{fwd}_{\mathrm{idom}(\varepsilon_2);\mathrm{idom}(\varepsilon_1)=\bot}\ (\mathrm{idom}(\mathring{\varepsilon}_2), \mathrm{idom}(\mathring{\varepsilon}_1)) = \mathring{\varepsilon}_{\bot 5} \\ \mathsf{fwd}_{\mathrm{icod}(\varepsilon_2);\mathrm{icod}(\varepsilon_2)=\varepsilon_6}\ (\mathrm{icod}(\mathring{\varepsilon}_1), \mathrm{icod}(\mathring{\varepsilon}_2)) = \mathring{\varepsilon}_6\end{array}}{\mathsf{fwd}_{\varepsilon_1;\varepsilon_2=\bot}\ (\mathring{\varepsilon}_1, \mathring{\varepsilon}_2) = \mathring{\varepsilon}_{\bot 5}}$$

$$\text{Fun-Err-r}\ \frac{\begin{array}{c}\varepsilon_1 \neq \langle ?, ? \rangle \text{ or } \varepsilon_2 \neq \langle ?, ? \rangle \\ \mathsf{fwd}_{\mathrm{idom}(\varepsilon_2);\mathrm{idom}(\varepsilon_1)=\varepsilon_5}\ (\mathrm{idom}(\mathring{\varepsilon}_2), \mathrm{idom}(\mathring{\varepsilon}_1)) = \mathring{\varepsilon}_5 \\ \mathsf{fwd}_{\mathrm{icod}(\varepsilon_2);\mathrm{icod}(\varepsilon_2)=\bot}\ (\mathrm{icod}(\mathring{\varepsilon}_1), \mathrm{icod}(\mathring{\varepsilon}_2)) = \mathring{\varepsilon}_{\bot 6}\end{array}}{\mathsf{fwd}_{\varepsilon_1;\varepsilon_2=\bot}\ (\mathring{\varepsilon}_1, \mathring{\varepsilon}_2) = \mathring{\varepsilon}_{\bot 6}}$$

$$\text{Fun-Pass}\ \frac{\begin{array}{c}\varepsilon_1 \neq \langle ?, ? \rangle \text{ or } \varepsilon_2 \neq \langle ?, ? \rangle \\ \mathsf{fwd}_{\mathrm{idom}(\varepsilon_2);\mathrm{idom}(\varepsilon_1)=\varepsilon_5}\ (\mathrm{idom}(\mathring{\varepsilon}_2), \mathrm{idom}(\mathring{\varepsilon}_1)) = \mathring{\varepsilon}_5 \\ \mathsf{fwd}_{\mathrm{icod}(\varepsilon_2);\mathrm{icod}(\varepsilon_2)=\varepsilon_6}\ (\mathrm{icod}(\mathring{\varepsilon}_1), \mathrm{icod}(\mathring{\varepsilon}_2)) = \mathring{\varepsilon}_6\end{array}}{\mathsf{fwd}_{\varepsilon_1;\varepsilon_2=\varepsilon_5 \to \varepsilon_6}\ (\mathring{\varepsilon}_1, \mathring{\varepsilon}_2) = \mathring{\varepsilon}_5 \to \mathring{\varepsilon}_6}$$

Figure 5.7: Forward Slicing for Evidence Composition in AGTLC, part 2

Theorem 37 (Forward slicing for evidence composition is well behaved for meets and joins)**.**
For any trace $\overset{\mathcal{D}}{\varepsilon_1 ; \varepsilon_2 = \varepsilon_\bot}$ *the function* $\underset{\varepsilon_1;\varepsilon_2=\varepsilon_\bot}{\mathsf{fwd}}\ (\mathring{\varepsilon}_1, \mathring{\varepsilon}_2) = \mathring{\varepsilon}_\bot$ *is a forward slicing function well behaved for meets and joins. Thus it uniquely determines its backward slicing function.*

5.5 SLICING FOR THE RUNTIME LANGUAGE ARL

The ARL language introduced in Section 3.1.7 is the runtime language for AGTLC. Whenever possible, I reuse the slicing systems presented in Sections 5.3 and 5.4 and Chapter 4 when designing a slicing system for the ARL runtime language.

I begin by defining slices for the runtime terms in ARL with a similar approach to previous languages: I introduce a ■ slice marker for the parts of a program that are considered irrelevant, both for runtime terms and values:

Syntax

$$
\begin{array}{lcll}
\mathring{\mathbf{e}} & ::= & \mathbf{n} \mid \mathbf{x} \mid \lambda \mathbf{x}.\mathbf{e} \mid \mathring{\varepsilon}\mathring{\mathbf{e}}\ \mathring{\varepsilon}\mathring{\mathbf{e}} \mid \mathring{\varepsilon}\mathring{\mathbf{e}} \mid \blacksquare & \text{(runtime term slices)} \\
\mathring{\mathbf{u}} & ::= & \mathbf{n} \mid \lambda \mathbf{x}.\mathring{\mathbf{e}} \mid \blacksquare & \text{(raw value slices)} \\
\mathring{\mathbf{v}} & ::= & \mathring{\mathbf{u}} \mid \mathring{\varepsilon}\mathring{\mathbf{u}} & \text{(value slices)}
\end{array}
$$

With slices defined, one can move to defining forward slicing for the runtime semantics. I follow a compositional approach, reusing slicing systems wherever the relations specifying the runtime semantics depend on previous definitions. I begin by providing a slicing system for the notions of reduction and then use this slicing system in the definition of forward slicing for contextual reduction.

5.5.1 *Forward slicing the notions of reduction*

ARL has only one notion of reduction: function application or beta reduction. This notion of reduction is different from the function applications presented in Chapter 4, in that it must also propagate evidence through the application. This propagation relies on evidence composition, which may fail and trigger a run-time type error. The notion of reduction for AGTLC function application discussed in Section 3.1.7 can be alternatively presented in a form that explicitly references the traces of evidence composition involved in every possible output:

$$
\text{Beta-Succ}\ \frac{\varepsilon_2 \fatsemi \mathit{idom}(\varepsilon_1) = \varepsilon_{\mathbf{u}} \qquad \big([(\varepsilon_{\mathbf{u}}\mathbf{u})/\mathbf{x}]\mathbf{e}\big) = \mathbf{e}'}{\varepsilon_1(\lambda \mathbf{x}.\mathbf{e})\ \varepsilon_2\mathbf{u} \rightsquigarrow \big(\mathit{icod}(\varepsilon_1)\big)\mathbf{e}'} \qquad \text{Beta-Fail}\ \frac{\varepsilon_2 \fatsemi \mathit{idom}(\varepsilon_1) = \bot}{\varepsilon_1(\lambda \mathbf{x}.\mathbf{e})\ \varepsilon_2\mathbf{u} \rightsquigarrow \mathbf{error}}
$$

To forward slice these notions of reduction, I can reuse the slicing systems both for evidence composition and for substitution discussed in Sections 4.4.1 and 5.4.

Figure 5.8 presents forward slicing for these two notions of reduction. Each of the original rules corresponds to three forward slicing rules, dealing with all possible slices that could be taken of the input and their impact on evaluation. Rules [Beta-Fail-Keep] and [Beta-Succ-Keep] reproduce the structure of each of the original rules in the notion of

$$\text{Beta-Fail-Keep}\ \frac{\underset{\varepsilon_2;idom(\varepsilon_1)=\bot}{\mathsf{fwd}}\ (\mathring{\varepsilon}_2, idom(\mathring{\varepsilon}_1)) = \bot \qquad \mathring{\mathbf{e}}_1 \sqsubseteq \lambda\mathbf{x}.\mathbf{e}}{\underset{\varepsilon_1(\lambda\mathbf{x}.\mathbf{e})\ \varepsilon_2\mathbf{u}\rightsquigarrow\mathbf{error}}{\mathsf{fwd}}\ (\mathring{\varepsilon}_1\mathring{\mathbf{e}}_1\ \mathring{\varepsilon}_2\mathring{\mathbf{u}}) = \mathbf{error}}$$

$$\text{Beta-Fail-Drop-Comp}\ \frac{\underset{\varepsilon_2;idom(\varepsilon_1)=\bot}{\mathsf{fwd}}\ (\mathring{\varepsilon}_2, idom(\mathring{\varepsilon}_1)) = \blacksquare \qquad \mathring{\mathbf{e}}_1 \sqsubseteq \lambda\mathbf{x}.\mathbf{e}}{\underset{\varepsilon_1(\lambda\mathbf{x}.\mathbf{e})\ \varepsilon_2\mathbf{u}\rightsquigarrow\mathbf{error}}{\mathsf{fwd}}\ (\mathring{\varepsilon}_1\mathring{\mathbf{e}}_1\ \mathring{\varepsilon}_2\mathring{\mathbf{u}}) = \blacksquare}$$

$$\text{Beta-Fail-Drop-All}\ \frac{}{\underset{\varepsilon_1(\lambda\mathbf{x}.\mathbf{e})\ \varepsilon_2\mathbf{u}\rightsquigarrow\mathbf{error}}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare}$$

$$\text{Beta-Succ-Keep}\ \frac{\underset{\varepsilon_2;idom(\varepsilon_1)=\varepsilon_{\mathbf{u}}}{\mathsf{fwd}}\ (\mathring{\varepsilon}_2, idom(\mathring{\varepsilon}_1)) = \mathring{\varepsilon}_{\mathbf{u}} \qquad \underset{[\varepsilon_{\mathbf{u}}\mathbf{u}/\mathbf{x}]\mathbf{e}=\mathbf{e}'}{[\mathring{\varepsilon}_{\mathbf{u}}\mathring{\mathbf{u}}/\mathbf{x}]\mathring{\mathbf{e}}} = \mathring{\mathbf{e}}'}{\underset{\varepsilon_1(\lambda\mathbf{x}.\mathbf{e})\ \varepsilon_2\mathbf{u}\rightsquigarrow icod(\varepsilon_1)\mathbf{e}'}{\mathsf{fwd}}\ (\mathring{\varepsilon}_1(\lambda\mathbf{x}.\mathring{\mathbf{e}})\ \mathring{\varepsilon}_2\mathring{\mathbf{u}}) = icod(\mathring{\varepsilon}_1)\mathring{\mathbf{e}}'}$$

$$\text{Beta-Succ-Drop-Lam-1}\ \frac{icod(\mathring{\varepsilon}_1) \neq \blacksquare}{\underset{\varepsilon_1(\lambda\mathbf{x}.\mathbf{e})\ \varepsilon_2\mathbf{u}\rightsquigarrow icod(\varepsilon_1)\mathbf{e}'}{\mathsf{fwd}}\ (\mathring{\varepsilon}_1\blacksquare\ \mathring{\varepsilon}_2\mathring{\mathbf{u}}) = icod(\mathring{\varepsilon}_1)\blacksquare}$$

$$\text{Beta-Succ-Drop-Lam-2}\ \frac{icod(\mathring{\varepsilon}_1) = \blacksquare}{\underset{\varepsilon_1(\lambda\mathbf{x}.\mathbf{e})\ \varepsilon_2\mathbf{u}\rightsquigarrow icod(\varepsilon_1)\mathbf{e}'}{\mathsf{fwd}}\ (\mathring{\varepsilon}_1\blacksquare\ \mathring{\varepsilon}_2\mathring{\mathbf{u}}) = \blacksquare}$$

$$\text{Beta-Succ-Drop-All}\ \frac{}{\underset{\varepsilon_1(\lambda\mathbf{x}.\mathbf{e})\ \varepsilon_2\mathbf{u}\rightsquigarrow icod(\varepsilon_1)\mathbf{e}'}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare}$$

Figure 5.8: Forward slicing for the notions of reduction in ARL.

reduction, proceeding when sufficient information to replicate behaviour is present: when evidence composition fails, the only information relevant is the failure, thus [Beta-Fail-Keep] focuses on the evidence propagation failure no matter the slices selected for the function or the arguments in the application. When evidence composition succeeds, I can rely on forward slicing for substitution, with minor modifications from the description in Chapter 4, to provide rule [Beta-Succ-Keep].

When the whole function application is marked as irrelevant, what happens with the evidence composition is implicitly also irrelevant, thus both rules [Beta-Fail-Drop-All] and [Beta-Succ-Drop-All] produce ■ as their output. The other rules deal with particular cases where a key portion of the original reduction is sliced out. Rule [Beta-Fail-Drop-Comp] produces an ■ when the evidence composition that should trigger the failure at hand produces ■. Rules [Beta-Succ-Drop-Lam-*] discard the application with an ■ when the applied function has been marked as ■, in which case the substitution has nothing to contribute. However, even though the application is discarded, it might be the case that the evidence information for the application result remains relevant. If that information is relevant, rule [Beta-Succ-Drop-Lam-1] preserves it. If that information is not relevant, rule [Beta-Succ-Drop-Lam-2] discards it and produces ■ for the final resulting slice.

These rules for forward slicing satisfy the requirements previously imposed.

Theorem 38 (Notions of reduction have forward slicing)**.**
$\mathsf{fwd}_{\mathbf{e} \rightsquigarrow \mathbf{e}'}(\mathring{\mathbf{e}}) = \mathring{\mathbf{e}}'$ *is well behaved for meets and joins, and is a forward slicing function. Thus it uniquely determines its backward slicing function.*

5.5.2 *Forward slicing contextual reduction*

ARL is defined as a reduction semantics, so I follow a similar structure to that in Section 4.6 to introduce slicing for the contextual reduction of ARL.

Slicing contextual reduction requires an inversion principle for evaluation contexts, ensuring that forward slicing is defined for all possible slices. Although the definition of evaluation contexts in ARL (and more generally in AGT languages) looks a bit different, it is possible to recover an equivalent inversion principle for slices to guarantee I have addressed the complete set of possible slices in the forward slicing rules. As a reminder, the definition of evaluation contexts in ARL is as follows:

$$\begin{array}{rcll} E \in \textsc{ECtxt} & ::= & \square \mid E[F[\varepsilon\square]] & \text{(evaluation contexts)} \\ F \in \textsc{EvFrame} & ::= & \square \mid \square\ \varepsilon\mathbf{e} \mid \varepsilon\mathbf{u}\ \square & \text{(evidence frame)} \end{array}$$

the difference from previous approaches is that every frame includes an evidence ascription. This subtle difference does not change the structure of the inversion principle, as evaluation contexts can still be split at any point into two sub contexts:

Lemma 7 (Inversion principle for slices of an AGT evaluation context)**.** *Let* $\mathring{t} \sqsubseteq E[t]$*. Then either*

- *(Slice preserves stack depth decomposition) There exists some* $\mathring{E} \sqsubseteq E$ *and some* $\mathring{t}' \sqsubseteq t$*, such that* $\mathring{t} = \mathring{E}[\mathring{t}']$*.*
- *(Irrelevance may drop frames) There exist some partition of the stack into two portions* E_1 *and* E_2 *such that* $E = E_1[E_2]$*, and there exists some* $\mathring{E} \sqsubseteq E_1$ *such that,* $\mathring{t} = \mathring{E}[\blacksquare]$*.*

I apply this inversion principle to guarantee all possible slices are handled when forward slicing contextual reduction traces. I introduce the forward slicing rules that are derived from each contextual reduction rule separately. Because all rules depend on other judgments for which I have already presented a slicing system, these can be reused. The structure of the reduction relation makes it impossible to justify these forward slicing rules using the diagrams I introduced earlier, but I discuss how each new rule is justified in terms of their dependencies on other slicing systems previously discussed.

- Contextual reduction takes the form $\dfrac{\mathbf{e} \rightsquigarrow \mathbf{e}'}{E[\mathbf{e}] \longrightarrow E[\mathbf{e}']}$.

  The trace of contextual reduction includes a trace of the computation of the notion of reduction, so one can defer to the forward slicing defined in Section 5.5.1. From the inversion principle one also needs to consider the case when slicing drops frames in the evaluation stack. In this case there is no information about the reduction at

this step, so forward slicing produces ■ plugged into the appropriate evaluation context. This produces two forward slicing rules:

$$\frac{\underset{\mathbf{e}\rightsquigarrow\mathbf{e}'}{\mathsf{fwd}}(\mathring{\mathbf{e}}) = \mathring{\mathbf{e}}' \qquad \mathring{E} \sqsubseteq E}{\underset{E[\mathbf{e}]\longmapsto E[\mathbf{e}']}{\mathsf{fwd}}\left(\mathring{E}[\mathring{\mathbf{e}}]\right) = \mathring{E}[\mathring{\mathbf{e}}']} \qquad \frac{\underset{\mathbf{e}\rightsquigarrow\mathbf{e}'}{\mathsf{fwd}}(\blacksquare) = \blacksquare \qquad \mathring{E}_1 \sqsubseteq E_1}{\underset{E_1[E_2[\mathbf{e}]]\longmapsto E_1[E_2[\mathbf{e}']]}{\mathsf{fwd}}\left(\mathring{E}_1[\blacksquare]\right) = \mathring{E}_1[\blacksquare]}$$

- Contextual reduction takes the form $\dfrac{\mathbf{e} \rightsquigarrow \mathbf{error}}{E[\mathbf{e}] \longrightarrow \mathbf{error}}$.

  Like in the previous case, forward slicing can refer to the forward slicing of notions of reduction for the failure-producing trace in the premise. I introduce three rules: two deal with the possible outputs of the notion of reduction, and the last rule deals with input slices that drop frames as in the previous case.

$$\frac{\underset{\mathbf{e}\rightsquigarrow\mathbf{error}}{\mathsf{fwd}}(\mathring{\mathbf{e}}) = \mathbf{error}}{\underset{E[\mathbf{e}]\longmapsto\mathbf{error}}{\mathsf{fwd}}\left(\mathring{E}[\mathring{\mathbf{e}}]\right) = \mathbf{error}} \qquad \frac{\underset{\mathbf{e}\rightsquigarrow\mathbf{error}}{\mathsf{fwd}}(\mathring{\mathbf{e}}) = \blacksquare}{\underset{E[\mathbf{e}]\longmapsto\mathbf{error}}{\mathsf{fwd}}\left(\mathring{E}[\mathring{\mathbf{e}}]\right) = \blacksquare}$$

$$\frac{\underset{\mathbf{e}\rightsquigarrow\mathbf{error}}{\mathsf{fwd}}(\blacksquare) = \blacksquare \qquad \mathring{E}_1 \sqsubseteq E_1}{\underset{E_1[E_2[\mathbf{e}]]\longmapsto\mathbf{error}}{\mathsf{fwd}}\left(\mathring{E}_1[\blacksquare]\right) = \blacksquare}$$

- Contextual reduction takes the form $\dfrac{\varepsilon_2 \fatsemi \varepsilon_1 = \bot}{E[F[\varepsilon_1\varepsilon_2\mathbf{u}]] \longrightarrow \mathbf{error}}$.

  This rule and the next deal with values wrapped on two evidence ascriptions and the propagation of their information. In this particular case, the evidence composition failed. This trace introduces four forward slicing rules to deal with all possible slices of $E[F[\varepsilon_1\varepsilon_2\mathbf{u}]]$ and of $\bot$. The first two rules deal with the trace of evidence composition and whether its slicing preserves the failure in its output slice or not. The third rule deals with the case when the innermost evidence ascription is dropped as irrelevant. The last rule deals with input slices that drop frames.

$$\frac{\underset{\varepsilon_2;\varepsilon_1=\bot}{\mathsf{fwd}}(\mathring{\varepsilon}_2, \mathring{\varepsilon}_1) = \bot \qquad \mathring{E} \sqsubseteq E \qquad \mathring{F} \sqsubseteq F}{\underset{E[F[\varepsilon_1\varepsilon_2\mathbf{u}]]\longmapsto\mathbf{error}}{\mathsf{fwd}}\left(\mathring{E}[\mathring{F}[\mathring{\varepsilon}_1\mathring{\varepsilon}_2\mathring{\mathbf{u}}]]\right) = \mathbf{error}} \qquad \frac{\underset{\varepsilon_2;\varepsilon_1=\bot}{\mathsf{fwd}}(\mathring{\varepsilon}_2, \mathring{\varepsilon}_1) = \blacksquare \qquad \mathring{E} \sqsubseteq E \qquad \mathring{F} \sqsubseteq F}{\underset{E[F[\varepsilon_1\varepsilon_2\mathbf{u}]]\longmapsto\mathbf{error}}{\mathsf{fwd}}\left(\mathring{E}[\mathring{F}[\mathring{\varepsilon}_1\mathring{\varepsilon}_2\mathring{\mathbf{u}}]]\right) = \blacksquare}$$

$$\frac{\underset{\varepsilon_2;\varepsilon_1=\bot}{\mathsf{fwd}}(\blacksquare, \mathring{\varepsilon}_1) = \blacksquare \qquad \mathring{E} \sqsubseteq E \qquad \mathring{F} \sqsubseteq F}{\underset{E[F[\varepsilon_1\varepsilon_2\mathbf{u}]]\longmapsto\mathbf{error}}{\mathsf{fwd}}\left(\mathring{E}[\mathring{F}[\mathring{\varepsilon}_1\blacksquare]]\right) = \blacksquare} \qquad \frac{\underset{\varepsilon_2;\varepsilon_1=\bot}{\mathsf{fwd}}(\blacksquare, \blacksquare) = \blacksquare \qquad \mathring{E}_1 \sqsubseteq E_1}{\underset{E_1[E_2[F[\varepsilon_1\varepsilon_2\mathbf{u}]]]\longmapsto\mathbf{error}}{\mathsf{fwd}}\left(\mathring{E}_1[\blacksquare]\right) = \blacksquare}$$

- Contextual reduction takes the form $\dfrac{}{E[F[\varepsilon_1\varepsilon_2\mathbf{u}]] \longrightarrow E[F[(\varepsilon_2 \fatsemi \varepsilon_1)\mathbf{u}]]}$.

In this case, the evidence composition succeeded. I need only three rules, depending on the slices one could possibly take from the input term $E[F[\varepsilon_1\varepsilon_2\mathbf{u}]]$. The first one deals with slices that preserve sufficient structure to replicate the reduction in the trace. The second and third ones are analogous to the third and fourth rules of the case where composition fails: they propagate the information available, which is bound to be insufficient to preserve the complete term, depending on whether the innermost evidence ascription has been marked as irrelevant (second rule) or if the slice selected dropped frames from the evaluation context (third rule).

$$\frac{\underset{\varepsilon_2;\varepsilon_1=\varepsilon_3}{\mathsf{fwd}}\ (\mathring{\varepsilon}_2, \mathring{\varepsilon}_1) = \mathring{\varepsilon}_3}{\underset{E[F[\varepsilon_1\varepsilon_2\mathbf{u}]]\longmapsto E[F[\varepsilon_3\mathbf{u}]]}{\mathsf{fwd}}\ \left(\mathring{E}[\mathring{F}[\mathring{\varepsilon}_1\mathring{\varepsilon}_2\mathring{\mathbf{u}}]]\right) = \mathring{E}[\mathring{F}[\mathring{\varepsilon}_3\mathring{\mathbf{u}}]]}$$

$$\frac{\underset{\varepsilon_2;\varepsilon_1=\varepsilon_3}{\mathsf{fwd}}\ (\blacksquare, \mathring{\varepsilon}_1) = \blacksquare}{\underset{E[F[\varepsilon_1\varepsilon_2\mathbf{u}]]\longmapsto E[F[\varepsilon_3\mathbf{u}]]}{\mathsf{fwd}}\ \left(\mathring{E}[\mathring{F}[\mathring{\varepsilon}_1\blacksquare]]\right) = \mathring{E}[\mathring{F}[\blacksquare\blacksquare]]}$$

$$\frac{\underset{\varepsilon_2;\varepsilon_1=\varepsilon_3}{\mathsf{fwd}}\ (\blacksquare, \blacksquare) = \blacksquare \qquad \mathring{E}_1 \sqsubseteq E_1}{\underset{E_1[E_2[F[\varepsilon_1\varepsilon_2\mathbf{u}]]]\longmapsto E_1[E_2[F[\varepsilon_3\mathbf{u}]]]}{\mathsf{fwd}}\ \left(\mathring{E}_1[\blacksquare]\right) = \mathring{E}_1[\blacksquare]}$$

These rules suffice to provide forward slicing.

Theorem 39 (Contextual reduction forward slicing is well behaved for meets and joins)**.** *For any trace* $\mathbf{e} \overset{\mathcal{D}}{\longmapsto} \mathbf{e}'$*, the function* $\underset{\mathbf{e}\longmapsto\mathbf{e}'}{\mathsf{fwd}}\ (\mathring{\mathbf{e}}) = \mathring{\mathbf{e}}'$ *is a forward slicing function well behaved for meets and joins. Thus it uniquely determines its backward slicing function.*

5.6 SLICING FOR BOUNDED ROWS AND RECORDS

The GTFL$_{\lesssim}$ language extends AGTLC with records and subtyping. In Section 3.4, I discuss an alternative run-time abstraction for gradual record types called Bounded Rows and Records (or BRR), which provides precise enforcement of type invariants. I now discuss a slicing system for GTFL$_{\lesssim}$ with the BRR run-time abstraction.

Thanks to the way I have structured consistent equality in AGTLC, providing slicing for the language with Bounded Rows and Records can build on the slicing system for AGTLC. All dependencies on consistent equality in AGTLC have been presented in a way compatible with the consistent subtyping of GTFL$_{\lesssim}$, so most of the work to reuse the rules in AGTLC can be accomplished by replacing references to $\sim$ with $\lesssim$.

But slicing for BRR still requires extending the definitions of the slicing systems in AGTLC to deal with new types, evidence objects, and expressions. The key novelty to provide slicing for this language manifests around slicing initial evidence and evidence composition, where most of the extra complexity of bounded rows and records resides. I focus on the initial evidence and evidence compositions slicing subsystems in this section.

5.6.1 *Slicing initial evidence*

As usual, the first step to provide slicing is describing slices for the inputs and output of the operation at hand. For initial evidence, one needs slices of gradual types and of evidence objects, which I provide using the same approach as before: introducing a new ■ slice and allowing slicing at every sub-part of the original syntactic structure.

$$
\begin{array}{rcll}
\mathring{S} \in \textsc{GTypeSl} & ::= & \mathsf{Int} \mid ? \mid \mathring{S} \to \mathring{S} \mid \blacksquare & \text{(gradual type slices)} \\
 & & \left| \left[\sum_{i=1}^{n} \ell_i : \mathring{M} \cdot \right] \right| \left[\sum_{i=1}^{n} \ell_i : \mathring{M}\ ? \right] & \\
\mathring{M} \in \textsc{MappingSl} & ::= & \varnothing \mid \mathring{S}_R \mid \mathring{S}_O \mid \blacksquare & \text{(mapping slices)} \\
\mathring{\varepsilon} \in \textsc{EvSl} & ::= & \langle \mathring{S}, \mathring{S} \rangle & \text{(evidence slices)} \\
\mathring{\varepsilon}_{\bot} \in \textsc{TotEvSl} & ::= & \mathring{\varepsilon} \mid \bot \mid \blacksquare & \text{(total evidence slices)}
\end{array}
$$

Note that I do not provide an inductive definition of evidence objects in this language. For AGTLC I proposed a definition of evidence objects that, while equivalent to a pair of gradual types, could be more easily used to inductively define evidence composition. I present evidence as pairs of gradual types as in Chapter 3. Extra constraints are captured by a well-formedness judgment.

The presence of Bounded Rows and Records in the types complicates the definition of evidence objects, introducing subtleties that reduce the advantages of inductively defining evidence objects. When an evidence object has an optional label on the right, it must combine both the possibility of depth record subtyping for the field when present on both sides (and thus inductively depend on a nested evidence object) with the possibility of width record subtyping when the field is absent on the right. When the field is absent on the right, the constraints on the left side are weaker than when the field is present on both sides. This challenge manifests in the following rule for well-formed evidence objects in Figure 3.13:

$$
\frac{S_1 \sqsubseteq S_3 \qquad \vdash \langle S_1, S_2 \rangle \ \mathbf{wf}}{\vdash \langle (S_3)_*, (S_2)_O \rangle \ \mathbf{wf}}
$$

Note that the type on the left of the evidence in the conclusion is not necessarily the same as the one in the premise.

Constraints like this one complicate building a definition of evidence objects with a direct inductive structure as I did for AGTLC in Section 5.4. For the interested reader, I include an inductive definition of evidence objects and evidence composition that follows an inductive structure in Appendix D. I do not follow that definition here as it provides no advantage.

5.6.1.1 *Assuming equal domains for gradual rows and records*

The presentation of both initial evidence and evidence composition can be simplified by assuming that the input record and row types (and evidence objects) include exactly the same labels. That is, for every individual label, either all the inputs have a mapping (See Section 3.4.3) for it or none does. This is not always the case in practice, but bounded rows

and records are expressive enough that whenever two row/record types (or evidence objects) do not have the same labels, there is a way to equivalently represent the types (or evidence objects) so that they both include exactly the same labels. One can first switch how the rows/records are represented to a syntax where they have the same labels by extracting the default mappings for every label that is missing in the other type.

By extracting from defaults, one obtains evidence object representations that have the same labels. This approach scales to deal with two evidence objects being composed, which can each now be assumed to have mappings for each domain, depending on the row designator defaults. That is, from a $\langle \left[\sum_{i=1}^{n} \ell_i : M_{i1}\ *_1\right], \left[\sum_{i=1}^{n} \ell_i : M_{i2}\ *_2\right] \rangle$ **wf** evidence object, one can extract a default mapping $\langle D(*_1), D(*_2) \rangle$ **wf** for any label mapping that is missing in the other evidence object. The default function D is defined in Figure 3.13.

5.6.1.2 *Total definition of initial evidence*

Assuming same-labelled inputs decreases the number of rules needed to define initial evidence and evidence composition for bounded rows and records. The rules I present in Figure 5.9 represent iterating through the (now shared) domain of mappings in both types, following a map-reduce (or *fold*) strategy. As presented, the computation of initial evidence for bounded rows and records preserves the structure of all possible successes and failures and ensures that applying forward slicing preserves all failing cases.

What does each of the rules in Figure 5.9 do? Rules [I-Dyn-RR] and [I-RR-Dyn] deal with a ? type on either input, by deferring to calculating initial evidence with the default information in a gradual row (after equalizing domains). The rest of the rules encode the process of iterating through the domain of mappings in a pair of types. Rules [I-Row-L-mt] and [I-Rec-L-mt] deal with all the cases that have no mappings. Each of the rest of the rules deals with a single step of iteration through the domain of labels and all possible failures that can arise: rule [I-RR-succ] deals with a successful computation of initial evidence, rule [I-RR-fail-head] deals with the first failure in the iteration, and [I-RR-fail-tail] and [I-RR-fail-both] propagate an already failing computation of initial evidence.

5.6.1.3 *Forward slicing for initial evidence*

To define forward slicing for initial evidence, I take advantage again of the fact that all failures are made explicit in the derivation trace. I can follow the same approach as done for AGTLC, where a failure only forward slices to a failure (rather than slicing to missing information ■) when every single failing subcomputation is preserved. If any of them are lost, forward slicing produces ■, because otherwise it would be presenting some of the sub-failures as irrelevant to the general failure. This approach also leads backward slicing to preserve necessary and sufficient information related to the failures when the initial evidence computation fails, relevant when we use initial evidence as part of evidence composition in Section 5.6.2.

I follow the structure of the rules presented in Figure 5.9 to introduce forward slicing rules. The way I presented the computation of initial evidence is key to producing a forward slicing function that mostly follows the structure of forward slicing for AGTLC.

$$\boxed{\mathcal{I}[\![S \lesssim S]\!] : \textsc{GType} \times \textsc{GType} \longrightarrow \textsc{Ev}_{\bot}^{<:}}$$

$$\text{I-Dyn-RR}\ \frac{\mathcal{I}[\![\left[\sum_{i=1}^{n} \ell_i : ?_O\ ?\right] \lesssim \left[\sum_{i=1}^{n} \ell_i : M_i *_2\right]]\!] = \varepsilon}{\mathcal{I}[\![? \lesssim \left[\sum_{i=1}^{n} \ell_i : M_i *_2\right]]\!] = \varepsilon} \qquad \text{I-Row-L-mt}\ \frac{}{\mathcal{I}[\![[?], [*_2]]\!] = \langle [?], [*_2] \rangle}$$

$$\text{I-RR-Dyn}\ \frac{\mathcal{I}[\![\left[\sum_{i=1}^{n} \ell_i : M_i *_1\right] \lesssim \left[\sum_{i=1}^{n} \ell_i : ?_O\ ?\right]]\!] = \varepsilon}{\mathcal{I}[\![\left[\sum_{i=1}^{n} \ell_i : M_i *_1\right] \lesssim ?]\!] = \varepsilon} \qquad \text{I-Rec-L-mt}\ \frac{}{\mathcal{I}[\![[\cdot], [*_2]]\!] = \langle [\cdot], [\cdot] \rangle}$$

$$\text{I-RR-succ}\ \frac{\begin{array}{c}\mathcal{I}[\![M_{i1}, M_{i2}]\!] = \langle M_{i3}, M_{i4} \rangle \\ \mathcal{I}[\![\left[\sum_{i=2}^{n} \ell_i : M_{i1} *_1\right], \left[\sum_{i=2}^{n} \ell_i : M_{i2} *_3\right]]\!] = \langle \left[\sum_{i=2}^{n} \ell_i : M_{i3} *_3\right], \left[\sum_{i=2}^{n} \ell_i : M_{i4} *_4\right]\end{array}}{\mathcal{I}[\![\left[\sum_{i=1}^{n} \ell_i : M_{i1} *_1\right], \left[\sum_{i=1}^{n} \ell_i : M_{i2} *_2\right]]\!] = \langle \left[\sum_{i=1}^{n} \ell_i : M_{i3} *_3\right], \left[\sum_{i=1}^{n} \ell_i : M_{i4} *_4\right] \rangle}$$

$$\text{I-RR-fail-tail}\ \frac{\mathcal{I}[\![M_{i1}, M_{i2}]\!] = \langle M_{i3}, M_{i4} \rangle \qquad \mathcal{I}[\![\left[\sum_{i=2}^{n} \ell_i : M_{i1} *_1\right], \left[\sum_{i=2}^{n} \ell_i : M_{i2} *_2\right]]\!] = \bot}{\mathcal{I}[\![\left[\sum_{i=1}^{n} \ell_i : M_{i1} *_1\right], \left[\sum_{i=1}^{n} \ell_i : M_{i2} *_2\right]]\!] = \bot}$$

$$\text{I-RR-fail-head}\ \frac{\begin{array}{c}\mathcal{I}[\![M_{i1}, M_{i2}]\!] = \bot \\ \mathcal{I}[\![\left[\sum_{i=2}^{n} \ell_i : M_{i1} *_1\right], \left[\sum_{i=2}^{n} \ell_i : M_{i2} *_2\right]]\!] = \langle \left[\sum_{i=2}^{n} \ell_i : M_{i3} *_3\right], \left[\sum_{i=2}^{n} \ell_i : M_{i4} *_4\right] \rangle\end{array}}{\mathcal{I}[\![\left[\sum_{i=1}^{n} \ell_i : M_{i1} *_1\right], \left[\sum_{i=1}^{n} \ell_i : M_{i2} *_2\right]]\!] = \bot}$$

$$\text{I-RR-fail-both}\ \frac{\mathcal{I}[\![M_{i1}, M_{i2}]\!] = \bot \qquad \mathcal{I}[\![\left[\sum_{i=2}^{n} \ell_i : M_{i1} *_1\right], \left[\sum_{i=2}^{n} \ell_i : M_{i2} *_2\right]]\!] = \bot}{\mathcal{I}[\![\left[\sum_{i=1}^{n} \ell_i : M_{i1} *_1\right], \left[\sum_{i=1}^{n} \ell_i : M_{i2} *_2\right]]\!] = \bot}$$

$$\boxed{\mathcal{I}[\![M \lesssim M]\!] : \textsc{Mapping} \times \textsc{Mapping} \longrightarrow \textsc{EvMapping}_{\bot}}$$

$$\frac{}{\mathcal{I}[\![M \lesssim \varnothing]\!] = \langle M, \varnothing \rangle} \qquad \frac{\mathcal{I}[\![S_1 \lesssim S_2]\!] = \langle S_3, S_4 \rangle}{\mathcal{I}[\![S_{1_{*_1}} \lesssim S_{2R}]\!] = \langle S_{3R}, S_{4R} \rangle} \qquad \frac{\mathcal{I}[\![S_1 \lesssim S_2]\!] = \bot}{\mathcal{I}[\![S_{1_{*_1}} \lesssim S_{2R}]\!] = \bot}$$

$$\frac{}{\mathcal{I}[\![\varnothing \lesssim S_{2R}]\!] = \bot} \qquad \frac{}{\mathcal{I}[\![\varnothing \lesssim S_{2O}]\!] = \langle \varnothing, \varnothing \rangle} \qquad \frac{\mathcal{I}[\![S_1 \lesssim S_2]\!] = \langle S_3, S_4 \rangle}{\mathcal{I}[\![S_{1_{*_1}} \lesssim S_{2O}]\!] = \langle S_{1_{*_1}}, S_{4O} \rangle}$$

$$\frac{\mathcal{I}[\![S_1 \lesssim S_2]\!] = \bot}{\mathcal{I}[\![S_{1_{*_1}} \lesssim S_{2O}]\!] = \langle S_{1_{*_1}}, \varnothing \rangle}$$

Figure 5.9: Initial Evidence for Bounded Rows and Records (selection of relevant rules for Row and Record Types)

Rules [I-Dyn-RR] and [I-RR-Dyn] only transform the gradual type ? into a type with the proper structure, so one can rely on the inductive call to produce some evidence slice that can be reused.

$$\frac{\underset{\mathcal{I}[\![[?]\lesssim\left[\sum_{i=1}^{n}\ell_i{:}M_i\ *_2\right]]\!]=\varepsilon}{\mathsf{fwd}}\left([?],\mathring{S}_2\right)=\mathring{\varepsilon}}{\underset{\mathcal{I}[\![?\lesssim\left[\sum_{i=1}^{n}\ell_i{:}M_i\ *_2\right]]\!]=\varepsilon}{\mathsf{fwd}}\left(\mathring{S}_1,\mathring{S}_2\right)=\mathring{\varepsilon}} \qquad \frac{\underset{\mathcal{I}[\![\left[\sum_{i=1}^{n}\ell_i{:}M_i\ *_1\right]\lesssim[?]]\!]=\varepsilon}{\mathsf{fwd}}\left(\mathring{S}_1,[?]\right)=\mathring{\varepsilon}}{\underset{\mathcal{I}[\![\left[\sum_{i=1}^{n}\ell_i{:}M_i\ *_1\right]\lesssim?]\!]=\varepsilon}{\mathsf{fwd}}\left(\mathring{S}_1,\mathring{S}_2\right)=\mathring{\varepsilon}}$$

Rules [I-Row-L-mt] and [I-Row-R-mt] deal with the base cases with no label mappings and provide the base case for the iterating definitions that deal with rows and records. If the input slices preserve the structure of the input types, slicing produces the complete original output. When any of the inputs is sliced to ■, slicing produces a least evidence slice ($\langle\blacksquare,\blacksquare\rangle$):

$$\frac{}{\underset{\mathcal{I}[\![[?]\lesssim[*_2]]\!]=\langle[?],[*_2]\rangle}{\mathsf{fwd}}([?],[*_2])=\langle[?],[*_2]\rangle} \qquad \frac{\mathring{S}_1=\blacksquare \text{ or } \mathring{S}_2=\blacksquare}{\underset{\mathcal{I}[\![[?]\lesssim[*_2]]\!]=\langle[?],[*_2]\rangle}{\mathsf{fwd}}\left(\mathring{S}_1,\mathring{S}_2\right)=\langle\blacksquare,\blacksquare\rangle}$$

$$\frac{}{\underset{\mathcal{I}[\![[\cdot]\lesssim[*_2]]\!]=\langle[\cdot],[\cdot]\rangle}{\mathsf{fwd}}([\cdot],[*_2])=\langle[\cdot],[\cdot]\rangle} \qquad \frac{\mathring{S}_1=\blacksquare \text{ or } \mathring{S}_2=\blacksquare}{\underset{\mathcal{I}[\![[\cdot]\lesssim[*_2]]\!]=\langle[\cdot],[\cdot]\rangle}{\mathsf{fwd}}\left(\mathring{S}_1,\mathring{S}_2\right)=\langle\blacksquare,\blacksquare\rangle}$$

Rule [I-RR-succ] represents the case when every premise successfully produces evidence outputs. I generate two forward slicing rules for it, depending on whether one has sliced out the types at hand or not:

$$\frac{\begin{array}{r}\underset{\mathcal{I}[\![M_{i1}\lesssim M_{i2}]\!]=\langle M_{i3},M_{i4}\rangle}{\mathsf{fwd}}\left(\mathring{M}_{i1},\mathring{M}_{i2}\right)=\langle\mathring{M}_{i3},\mathring{M}_{i4}\rangle\\ \underset{\mathcal{I}[\![\left[\sum_{i=2}^{n}\ell_i{:}M_{i1}\ *_1\right]\lesssim\left[\sum_{i=2}^{n}\ell_i{:}M_{i2}\ *_3\right]]\!]=\left\langle\left[\sum_{i=2}^{n}\ell_i{:}M_{i3}\ *_3\right],\left[\sum_{i=2}^{n}\ell_i{:}M_{i4}\ *_4\right]\right\rangle}{\mathsf{fwd}}\left(\left[\sum_{i=2}^{n}\ell_i:\mathring{M}_{i1}\ *_1\right],\left[\sum_{i=2}^{n}\ell_i:\mathring{M}_{i2}\ *_3\right]\right)\\ =\left\langle\left[\sum_{i=2}^{n}\ell_i:\mathring{M}_{i3}\ *_3\right],\left[\sum_{i=2}^{n}\ell_i:\mathring{M}_{i4}\ *_4\right]\right\rangle\end{array}}{\begin{array}{r}\underset{\mathcal{I}[\![\left[\sum_{i=1}^{n}\ell_i{:}M_{i1}\ *_1\right]\lesssim\left[\sum_{i=1}^{n}\ell_i{:}M_{i2}\ *_2\right]]\!]=\langle\left[\sum_{i=1}^{n}\ell_i{:}M_{i3}\ *_3\right],\left[\sum_{i=1}^{n}\ell_i{:}M_{i4}\ *_4\right]\rangle}{\mathsf{fwd}}\left(\left[\sum_{i=1}^{n}\ell_i:\mathring{M}_{i1}\ *_1\right],\left[\sum_{i=1}^{n}\ell_i:\mathring{M}_{i2}\ *_2\right]\right)\\ =\left\langle\left[\sum_{i=1}^{n}\ell_i:\mathring{M}_{i3}\ *_3\right],\left[\sum_{i=1}^{n}\ell_i:\mathring{M}_{i4}\ *_4\right]\right\rangle\end{array}}$$

$$\frac{\mathring{S}_1=\blacksquare \text{ or } \mathring{S}_2=\blacksquare}{\underset{\mathcal{I}[\![\left[\sum_{i=1}^{n}\ell_i{:}M_{i1}\ *_1\right]\lesssim\left[\sum_{i=1}^{n}\ell_i{:}M_{i2}\ *_2\right]]\!]=\left\langle\left[\sum_{i=1}^{n}\ell_i{:}M_{i3}\ *_3\right],\left[\sum_{i=1}^{n}\ell_i{:}M_{i4}\ *_4\right]\right\rangle}{\mathsf{fwd}}\left(\mathring{S}_1,\mathring{S}_2\right)=\langle\blacksquare,\blacksquare\rangle}$$

The last three rules for initial evidence in Figure 5.9 deal with the structural propagation of failures. Rules [I-RR-fail-head] and [I-RR-fail-tail] deal with the cases where only one

of the premises produces an initial evidence failure, and they follow a similar structure that generates two forward slicing rules from each initial evidence rule.

$$\frac{\begin{array}{c}\overset{\mathsf{fwd}}{\mathcal{I}[\![M_{i1} \lesssim M_{i2}]\!] = \langle M_{i3}, M_{i4}\rangle}\left(\mathring{M}_{i1}, \mathring{M}_{i2}\right) = \langle \mathring{M}_{i3}, \mathring{M}_{i4}\rangle \\ \overset{\mathsf{fwd}}{\mathcal{I}[\![\left[\sum_{i=2}^{n} \ell_i : M_{i1} *_1\right] \lesssim \left[\sum_{i=2}^{n} \ell_i : M_{i2} *_2\right]]\!] = \bot}\left(\left[\sum_{i=2}^{n} \ell_i : \mathring{M}_{i1} *_1\right], \left[\sum_{i=2}^{n} \ell_i : \mathring{M}_{i2} *_2\right]\right) = \mathring{\varepsilon}_{\bot}\end{array}}{\overset{\mathsf{fwd}}{\mathcal{I}[\![\left[\sum_{i=1}^{n} \ell_i : M_{i1} *_1\right] \lesssim \left[\sum_{i=1}^{n} \ell_i : M_{i2} *_2\right]]\!] = \bot}\left(\left[\sum_{i=1}^{n} \ell_i : \mathring{M}_{i1} *_1\right], \left[\sum_{i=1}^{n} \ell_i : \mathring{M}_{i2} *_2\right]\right) = \mathring{\varepsilon}_{\bot}}$$

$$\frac{\mathring{S}_1 = \blacksquare \text{ or } \mathring{S}_2 = \blacksquare}{\overset{\mathsf{fwd}}{\mathcal{I}[\![\left[\sum_{i=1}^{n} \ell_i : M_{i1} *_1\right] \lesssim \left[\sum_{i=1}^{n} \ell_i : M_{i2} *_2\right]]\!] = \bot}\left(\mathring{S}_1, \mathring{S}_2\right) = \blacksquare}$$

$$\frac{\begin{array}{c}\overset{\mathsf{fwd}}{\mathcal{I}[\![M_{i1} \lesssim M_{i2}]\!] = \bot}\left(\mathring{M}_{i1}, \mathring{M}_{i2}\right) = \mathring{\varepsilon}_{\bot} \\ \overset{\mathsf{fwd}}{\mathcal{I}[\![\left[\sum_{i=2}^{n} \ell_i : M_{i1} *_1\right] \lesssim \left[\sum_{i=2}^{n} \ell_i : M_{i2} *_3\right]]\!] = \left\langle \left[\sum_{i=2}^{n} \ell_i : M_{i3} *_3\right], \left[\sum_{i=2}^{n} \ell_i : M_{i4} *_4\right]\right\rangle}\left(\left[\sum_{i=2}^{n} \ell_i : \mathring{M}_{i1} *_1\right], \left[\sum_{i=2}^{n} \ell_i : \mathring{M}_{i2} *_3\right]\right) \\ = \left\langle \left[\sum_{i=2}^{n} \ell_i : \mathring{M}_{i3} *_3\right], \left[\sum_{i=2}^{n} \ell_i : \mathring{M}_{i4} *_4\right]\right\rangle\end{array}}{\overset{\mathsf{fwd}}{\mathcal{I}[\![\left[\sum_{i=1}^{n} \ell_i : M_{i1} *_1\right] \lesssim \left[\sum_{i=1}^{n} \ell_i : M_{i2} *_2\right]]\!] = \bot}\left(\left[\sum_{i=1}^{n} \ell_i : \mathring{M}_{i1} *_1\right], \left[\sum_{i=1}^{n} \ell_i : \mathring{M}_{i2} *_2\right]\right) = \mathring{\varepsilon}_{\bot}}$$

$$\frac{\mathring{S}_1 = \blacksquare \text{ or } \mathring{S}_2 = \blacksquare}{\overset{\mathsf{fwd}}{\mathcal{I}[\![\left[\sum_{i=1}^{n} \ell_i : M_{i1} *_1\right] \lesssim \left[\sum_{i=1}^{n} \ell_i : M_{i2} *_2\right]]\!] = \bot}\left(\mathring{S}_1, \mathring{S}_2\right) = \blacksquare}$$

Finally, rule [I-RR-fail-both] deals with multiple failures in the premises. As justified when discussing AGTLC, I choose to only forward slice this failure to $\bot$ when forward

slicing both premises preserves their failure information, leading to the following two forward slicing rules:

$$\frac{\begin{array}{c}\mathsf{fwd}_{\mathcal{I}[\![M_{i1} \lesssim M_{i2}]\!]=\bot}\left(\mathring{M}_{i1}, \mathring{M}_{i2}\right) = \bot \\ \mathsf{fwd}_{\mathcal{I}[\![\left[\sum_{i=2}^{n} \ell_i : M_{i1} *_1\right] \lesssim \left[\sum_{i=2}^{n} \ell_i : M_{i2} *_2\right]]\!]=\bot}\left(\left[\sum_{i=2}^{n} \ell_i : \mathring{M}_{i1} *_1\right], \left[\sum_{i=2}^{n} \ell_i : \mathring{M}_{i2} *_2\right]\right) = \bot\end{array}}{\mathsf{fwd}_{\mathcal{I}[\![\left[\sum_{i=1}^{n} \ell_i : M_{i1} *_1\right] \lesssim \left[\sum_{i=1}^{n} \ell_i : M_{i2} *_2\right]]\!]=\bot}\left(\left[\sum_{i=1}^{n} \ell_i : \mathring{M}_{i1} *_1\right], \left[\sum_{i=1}^{n} \ell_i : \mathring{M}_{i2} *_2\right]\right) = \bot}$$

$$\frac{\begin{array}{c}\mathsf{fwd}_{\mathcal{I}[\![M_{i1} \lesssim M_{i2}]\!]=\bot}\left(\mathring{M}_{i1}, \mathring{M}_{i2}\right) = \varepsilon_{\bot 1} \\ \mathsf{fwd}_{\mathcal{I}[\![\left[\sum_{i=2}^{n} \ell_i : M_{i1} *_1\right] \lesssim \left[\sum_{i=2}^{n} \ell_i : M_{i2} *_2\right]]\!]=\bot}\left(\left[\sum_{i=2}^{n} \ell_i : \mathring{M}_{i1} *_1\right], \left[\sum_{i=2}^{n} \ell_i : \mathring{M}_{i2} *_2\right]\right) = \varepsilon_{\bot 2} \\ \varepsilon_{\bot 1} = \blacksquare \text{ or } \varepsilon_{\bot 2} = \blacksquare\end{array}}{\mathsf{fwd}_{\mathcal{I}[\![\left[\sum_{i=1}^{n} \ell_i : M_{i1} *_1\right] \lesssim \left[\sum_{i=1}^{n} \ell_i : M_{i2} *_2\right]]\!]=\bot}\left(\left[\sum_{i=1}^{n} \ell_i : \mathring{M}_{i1} *_1\right], \left[\sum_{i=1}^{n} \ell_i : \mathring{M}_{i2} *_2\right]\right) = \blacksquare}$$

I must still provide forward slicing for the rules defining initial evidence over mappings M. It is in the rules for mappings that most of the interesting forward slicing happens for bounded rows and records. I focus my attention on the rules that have a different structure from previous developments and how slicing interacts with the "optional" and "required" labels in mappings.

First, I address the failing rules for a required mapping on the supertype. Initial evidence rule

$$\frac{}{\mathcal{I}[\![\varnothing \lesssim S_{2R}]\!] = \bot}$$

triggers a failure because of restrictions introduced by record width subtyping: if a field is required in the supertype, it must be present in the subtype. This induces the following forward slicing rules, which preserve a failure when the mapping structure (a missing mapping on the left and a required mapping on the right) is present. Otherwise, forward slicing produces $\blacksquare$:

$$\frac{}{\mathsf{fwd}_{\mathcal{I}[\![\varnothing \lesssim S_{2_R}]\!]=\bot}\left(\varnothing, \mathring{S}_{2_R}\right) = \bot} \qquad \frac{\mathring{M}_1 = \blacksquare \text{ or } \mathring{M}_2 = \blacksquare}{\mathsf{fwd}_{\mathcal{I}[\![\varnothing \lesssim S_{2_R}]\!]=\mathring{M}_3}\left(\mathring{M}_1, \mathring{M}_2\right) = \blacksquare}$$

Rules induced from $\dfrac{\mathcal{I}[\![S_1 \lesssim S_2]\!] = \bot}{\mathcal{I}[\![S_{1*_1} \lesssim S_{2R}]\!] = \bot}$ depend on a failure in the premise, so I can rely on forward slicing of initial evidence for gradual types. If the premise preserves a failure, then I propagate the failure, otherwise slicing preserves no relevant information:

$$\frac{\underset{\mathcal{I}[\![S_1 \lesssim S_2]\!]=\bot}{\mathsf{fwd}}\left(\mathring{S}_1, \mathring{S}_2\right) = \bot}{\underset{\mathcal{I}[\![S_{1_{*_1}} \lesssim S_{2_R}]\!]=\bot}{\mathsf{fwd}}\left(\mathring{S}_{1_{*_1}}, \mathring{S}_{2_R}\right) = \bot} \qquad \frac{\underset{\mathcal{I}[\![S_1 \lesssim S_2]\!]=\bot}{\mathsf{fwd}}\left(\mathring{S}_1, \mathring{S}_2\right) = \blacksquare}{\underset{\mathcal{I}[\![S_{1_{*_1}} \lesssim S_{2_R}]\!]=\bot}{\mathsf{fwd}}\left(\mathring{S}_{1_{*_1}}, \mathring{S}_{2_R}\right) = \blacksquare}$$

$$\frac{\mathring{M}_1 = \blacksquare \text{ or } \mathring{M}_2 = \blacksquare}{\underset{\mathcal{I}[\![S_{1_{*_1}} \lesssim S_{2_R}]\!]=\bot}{\mathsf{fwd}}\left(\mathring{M}_1, \mathring{M}_2\right) = \blacksquare}$$

Two other interesting rules are the rules that have the same mapping constraints on the subtype but an optional mapping on the supertype instead of a required mapping. These cases are interesting because the optionality interacts with width subtyping and abort the failure. Instead of failing as before, I can further constrain the types so that the supertype is forced to be missing.

First, consider the following initial evidence rule

$$\frac{}{\mathcal{I}[\![\varnothing \lesssim S_{2O}]\!] = \langle \varnothing, \varnothing \rangle}$$

If the slice keeps sufficient structure to observe that the supertype is an optional mapping, I preserve the successful constraining. If not, I mark the mappings as irrelevant as follows:

$$\frac{}{\underset{\mathcal{I}[\![\varnothing \lesssim S_{2_O}]\!]=\langle\varnothing,\varnothing\rangle}{\mathsf{fwd}}\left(\varnothing, \mathring{S}_{2_O}\right) = \langle \varnothing, \varnothing \rangle} \qquad \frac{\mathring{M}_1 = \blacksquare \text{ or } \mathring{M}_2 = \blacksquare}{\underset{\mathcal{I}[\![\varnothing \lesssim S_{2_O}]\!]=\langle\varnothing,\varnothing\rangle}{\mathsf{fwd}}\left(\mathring{M}_1, \mathring{M}_2\right) = \langle \blacksquare, \blacksquare \rangle}$$

Lastly, consider rule $\dfrac{\mathcal{I}[\![S_1 \lesssim S_2]\!] = \bot}{\mathcal{I}[\![S_{1_{*_1}} \lesssim S_{2O}]\!] = \langle S_{1_{*_1}}, \varnothing \rangle}$. In this rule, the premise has failed, but that does not imply a failure, instead it implies constraining the valid types again (the optional supertype must be missing). If slices preserve sufficient structure for the premise slicing to preserve the failure, then I also preserve the constraining outcome. If the input slices do not preserve sufficient structure, or if slices preserve sufficient structure but the slicing premise does not preserve the failure, then I mark the resulting evidence mapping as irrelevant:

$$\frac{\underset{\mathcal{I}[\![S_1 \lesssim S_2]\!]=\bot}{\mathsf{fwd}}\left(\mathring{S}_1, \mathring{S}_2\right) = \bot}{\underset{\mathcal{I}[\![S_{1_{*_1}} \lesssim S_{2_O}]\!]=\langle S_{1_{*_1}},\varnothing\rangle}{\mathsf{fwd}}\left(\mathring{S}_{1_{*_1}}, \mathring{S}_{2_O}\right) = \langle \mathring{S}_{1_{*_1}}, \varnothing \rangle}$$

$$\frac{\underset{\mathcal{I}[\![S_1 \lesssim S_2]\!]=\bot}{\mathsf{fwd}}\left(\mathring{S}_1, \mathring{S}_2\right) = \blacksquare}{\underset{\mathcal{I}[\![S_{1_{*_1}} \lesssim S_{2_O}]\!]=\langle S_{1_{*_1}},\varnothing\rangle}{\mathsf{fwd}}\left(\mathring{S}_{1_{*_1}}, \mathring{S}_{2_O}\right) = \langle \blacksquare, \blacksquare \rangle} \qquad \frac{\mathring{M}_1 = \blacksquare \text{ or } \mathring{M}_2 = \blacksquare}{\underset{\mathcal{I}[\![S_{1_{*_1}} \lesssim S_{2_O}]\!]=\langle S_{1_{*_1}},\varnothing\rangle}{\mathsf{fwd}}\left(\mathring{M}_1, \mathring{M}_2\right) = \langle \blacksquare, \blacksquare \rangle}$$

The definition of forward slicing satisfies the constraints to use it with the Galois slicing framework.

Theorem 40 (Forward slicing for Initial evidence with BRR is well behaved for meets and joins)**.**
For any trace $\overset{\mathcal{D}}{\mathcal{I}[\![S_1 \lesssim S_2]\!]}$ *the function* $\underset{\mathcal{I}[\![S_1 \lesssim S_2]\!]=\varepsilon_\perp}{\mathsf{fwd}}\left(\mathring{S}_1, \mathring{S}_2\right) = \mathring{\varepsilon}_\perp$ *is a forward slicing function well behaved for meets and joins. Thus it uniquely determines its backward slicing function.*

5.6.2 *Slicing evidence composition for BRR*

I take the opportunity of dealing with BRR to present a different approach to slicing evidence composition than the one I previously presented for AGTLC. For AGTLC, I applied Galois slicing to an inductive definition of evidence composition. One could also apply forward slicing to the definitions in Appendix D for GTFL$_{\lesssim}$. However, I take the opportunity to provide slicing to an alternative presentation of evidence composition used in AGT systems.

Evidence composition can be defined as a combination of the initial evidence and gradual meet operations. Since I discussed slicing for initial evidence in the previous section, I reuse that work by relying on the equivalence from Proposition 20 in Chapter 3. That definition of evidence composition must be extended from a partial function to a total function, as follows:

Proposition 41 (Evidence Composition in terms of other operations (Bañados Schwerter, Clark, et al., 2021))**.**
Evidence composition can be computed by relying on initial evidence operations and gradual meets:

$$\frac{\begin{array}{c}\pi_2(\varepsilon_1) \sqcap \pi_1(\varepsilon_2) = S_{mid} \\ \mathcal{I}[\![\pi_1(\varepsilon_1) \lesssim S_{mid}]\!] = \varepsilon_3 \quad \mathcal{I}[\![S_{mid} \lesssim \pi_2(\varepsilon_2)]\!] = \varepsilon_4 \\ \mathcal{I}[\![\pi_1(\varepsilon_3) \lesssim \pi_2(\varepsilon_4)]\!] = \varepsilon_5\end{array}}{\varepsilon_1 ; \varepsilon_2 = \varepsilon_5}$$

$\boxed{S \sqcap S = S_\bot}$

$$\frac{}{? \sqcap S_2 = S_2} \qquad \frac{}{S_1 \sqcap ? = S_1} \qquad \frac{}{\mathsf{Int} \sqcap \mathsf{Int} = \mathsf{Int}} \qquad \frac{}{\mathsf{Int} \sqcap S_1 \to S_2 = \bot}$$

$$\frac{}{S_1 \to S_2 \sqcap \mathsf{Int} = \bot} \qquad \frac{S_1 \sqcap S_3 = S_5 \quad S_2 \sqcap S_4 = S_6}{S_1 \to S_2 \sqcap S_3 \to S_4 = S_5 \to S_6}$$

$$\frac{S_1 \sqcap S_3 = \bot \quad S_2 \sqcap S_4 = S_6}{S_1 \to S_2 \sqcap S_3 \to S_4 = \bot} \qquad \frac{S_1 \sqcap S_3 = S_5 \quad S_2 \sqcap S_4 = \bot}{S_1 \to S_2 \sqcap S_3 \to S_4 = \bot}$$

$$\frac{S_1 \sqcap S_3 = \bot \quad S_2 \sqcap S_4 = \bot}{S_1 \to S_2 \sqcap S_3 \to S_4 = \bot} \qquad \frac{}{\mathsf{Int} \sqcap \left[\sum_{i=1}^{n} \ell_i : M_i *_2\right] = \bot}$$

$$\frac{}{S_1 \to S_2 \sqcap \left[\sum_{i=1}^{n} \ell_i : M_i *_2\right] = \bot} \qquad \frac{}{\left[\sum_{i=1}^{n} \ell_i : M_i *_1\right] \sqcap \mathsf{Int} = \bot}$$

$$\frac{}{\left[\sum_{i=1}^{n} \ell_i : M_i *_1\right] \sqcap S_1 \to S_2 = \bot} \qquad \frac{}{[\cdot] \sqcap [*_2] = [\cdot]} \qquad \frac{}{[*_1] \sqcap [\cdot] = [\cdot]}$$

$$\frac{}{[?] \sqcap [?] = [?]}$$

$$\frac{M_1 \sqcap M_2 = M_3 \qquad \left[\sum_{i=2}^{n} \ell_i : M_{i1} *_1\right] \sqcap \left[\sum_{i=2}^{n} \ell_i : M_{i2} *_2\right] = \left[\sum_{i=2}^{n} \ell_i : M_{i3} *_3\right]}{\left[\sum_{i=1}^{n} \ell_i : M_{i1} *_1\right] \sqcap \left[\sum_{i=1}^{n} \ell_i : M_{i2} *_2\right] = \left[\sum_{i=1}^{n} \ell_i : M_{i3} *_3\right]}$$

$$\frac{M_1 \sqcap M_2 = \bot \qquad \left[\sum_{i=2}^{n} \ell_i : M_{i1} *_1\right] \sqcap \left[\sum_{i=2}^{n} \ell_i : M_{i2} *_2\right] = \left[\sum_{i=2}^{n} \ell_i : M_{i3} *_3\right]}{\left[\sum_{i=1}^{n} \ell_i : M_{i1} *_1\right] \sqcap \left[\sum_{i=1}^{n} \ell_i : M_{i2} *_2\right] = \bot}$$

$$\frac{M_1 \sqcap M_2 = M_3 \qquad \left[\sum_{i=2}^{n} \ell_i : M_{i1} *_1\right] \sqcap \left[\sum_{i=2}^{n} \ell_i : M_{i2} *_2\right] = \bot}{\left[\sum_{i=1}^{n} \ell_i : M_{i1} *_1\right] \sqcap \left[\sum_{i=1}^{n} \ell_i : M_{i2} *_2\right] = \bot}$$

$$\frac{M_1 \sqcap M_2 = \bot \qquad \left[\sum_{i=2}^{n} \ell_i : M_{i1} *_1\right] \sqcap \left[\sum_{i=2}^{n} \ell_i : M_{i2} *_2\right] = \bot}{\left[\sum_{i=1}^{n} \ell_i : M_{i1} *_1\right] \sqcap \left[\sum_{i=1}^{n} \ell_i : M_{i2} *_2\right] = \bot}$$

Figure 5.10: Gradual Meet with BRR, part 1

$\boxed{M \sqcap M = M_\bot}$

$$\frac{M \neq S_R}{\varnothing \sqcap M = \varnothing} \qquad \frac{}{\varnothing \sqcap S_R = \bot} \qquad \frac{M \neq S_R}{M \sqcap \varnothing = \varnothing} \qquad \frac{}{S_R \sqcap \varnothing = \bot}$$

$$\frac{S_1 \sqcap S_2 = S_3}{S_{1O} \sqcap S_{2O} = S_{3O}} \qquad \frac{S_1 \sqcap S_2 = \bot}{S_{1O} \sqcap S_{2O} = \varnothing} \qquad \frac{S_1 \sqcap S_2 = S_3 \quad *_1 = R \text{ or } *_2 = R}{S_{1_{*_1}} \sqcap S_{2_{*_2}} = S_{3R}}$$

$$\frac{S_1 \sqcap S_2 = \bot \quad *_1 = R \text{ or } *_2 = R}{S_{1_{*_1}} \sqcap S_{2_{*_2}} = \bot}$$

Figure 5.11: Gradual Meet with BRR, part 2

To provide a total definition of evidence composition, all possible failures involved need to be accounted for, leading to the introduction of the following extra rules:

$$\frac{\pi_2(\varepsilon_1) \sqcap \pi_1(\varepsilon_2) = \bot}{\varepsilon_1 ; \varepsilon_2 = \bot} \qquad \frac{\begin{array}{c}\pi_2(\varepsilon_1) \sqcap \pi_1(\varepsilon_2) = S_{mid} \\ \mathcal{I}[\![\pi_1(\varepsilon_1) \lesssim S_{mid}]\!] = \bot \quad \mathcal{I}[\![S_{mid} \lesssim \pi_2(\varepsilon_2)]\!] = \varepsilon_4\end{array}}{\varepsilon_1 ; \varepsilon_2 = \bot}$$

$$\frac{\begin{array}{c}\pi_2(\varepsilon_1) \sqcap \pi_1(\varepsilon_2) = S_{mid} \\ \mathcal{I}[\![\pi_1(\varepsilon_1) \lesssim S_{mid}]\!] = \varepsilon_3 \quad \mathcal{I}[\![S_{mid} \lesssim \pi_2(\varepsilon_2)]\!] = \bot\end{array}}{\varepsilon_1 ; \varepsilon_2 = \bot}$$

$$\frac{\begin{array}{c}\pi_2(\varepsilon_1) \sqcap \pi_1(\varepsilon_2) = S_{mid} \\ \mathcal{I}[\![\pi_1(\varepsilon_1) \lesssim S_{mid}]\!] = \bot \quad \mathcal{I}[\![S_{mid} \lesssim \pi_2(\varepsilon_2)]\!] = \bot\end{array}}{\varepsilon_1 ; \varepsilon_2 = \bot}$$

$$\frac{\begin{array}{c}\pi_2(\varepsilon_1) \sqcap \pi_1(\varepsilon_2) = S_{mid} \\ \mathcal{I}[\![\pi_1(\varepsilon_1) \lesssim S_{mid}]\!] = \varepsilon_3 \quad \mathcal{I}[\![S_{mid} \lesssim \pi_2(\varepsilon_2)]\!] = \varepsilon_4 \\ \mathcal{I}[\![\pi_1(\varepsilon_3) \lesssim \pi_2(\varepsilon_4)]\!] = \bot\end{array}}{\varepsilon_1 ; \varepsilon_2 = \bot}$$

Using the equivalence in Proposition 41, slicing for evidence composition can be defined by introducing slicing for gradual meets and relying on previous definitions.

Another reason for this approach to slicing evidence composition in BRR is that the inductive definition of evidence composition in Appendix D relies on the gradual meet operation, and initial evidence is still required for slicing elaboration. Thus separately defining a forward slicing for evidence composition gains nothing: one would still need to provide slicing for gradual meets and initial evidence.

I introduce an inductive definition for gradual meets in Figures 5.10 and 5.11. The definition follows a similar structure to the total version of initial evidence I discuss in Section 5.3.1 but produces a single type instead of an evidence object. The gradual meet between two gradual types, $S_1 \sqcap S_2$, is defined as the greatest lower bound over the gradual precision ordering between both types, which can also be achieved by computing $\alpha(\gamma(S_1) \cap \gamma(S_2))$ when the intersection does not produce an empty set. When the intersection is empty, the meet fails. This specification is equivalent to these inductive rules.

The inductive rules for forward slicing of gradual meet are structurally similar to the definitions of forward slicing for initial evidence and can be derived mechanically following the same guidelines. I can now rely on the definition of evidence composition from Proposition 41 to introduce slicing for evidence composition with bounded rows and records in Figure 5.12.

Theorem 42 (Forward slicing for evidence composition with BRR is well behaved for meets and joins)**.**
For any trace $\underset{\varepsilon_1;\varepsilon_2=\varepsilon_\bot}{\mathcal{D}}$ *the function* $\underset{\varepsilon_1;\varepsilon_2=\varepsilon_\bot}{\mathsf{fwd}}(\mathring{\varepsilon}_1, \mathring{\varepsilon}_2) = \mathring{\varepsilon}_\bot$ *is a forward slicing function well behaved for meets and joins. Thus it uniquely determines its backward slicing function.*

This extended slicing system for evidence composition can be used to extend forward and backward slicing to the new program traces in the language. Whenever a program trace ends in error, backward slicing can be used to obtain the parts of the original program involved in the failure.

5.6.3 *A slicing example for BRR*

Now that I have presented a full slicing system, let's revisit the example from Section 5.1:

$$\begin{array}{l}\mathsf{let}\ \mathit{input} : ? := [\mathit{transform} : 2, \mathit{size} : \lambda(x : \mathsf{Int}).x]\ \mathsf{in}\\ \quad \mathsf{let}\ \mathit{config} : [\mathit{size} : \mathsf{Int}, \mathit{transform} : \mathsf{Int} \to \mathsf{Int}] := \mathit{input}\ \mathsf{in}\\ \quad\quad (\mathit{config}.\mathit{transform})\ (\mathit{config}.\mathit{size})\end{array}$$

How does this program run to a failure and what information does slicing produce for it? Assuming that the let bindings desugar to function applications, this program elaborates to the following runtime program:

$$\begin{array}{l}\varepsilon_1\,(\lambda\,\mathbf{input}\,.\,(\varepsilon_3\,(\lambda\,\mathbf{config}\,.\,(\varepsilon_5\,(\varepsilon_7\,\mathbf{config}\,.\,\mathit{transform})\ \ \varepsilon_6\,(\varepsilon_8\,\mathbf{config}\,.\,\mathit{size})))\,\varepsilon_4\,\mathbf{input}))\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \varepsilon_2[\mathit{transform} = 2, \mathit{size} = \lambda\mathbf{x}\,.\,\mathbf{x}]\end{array}$$

where the line split separates a function application. I have hidden the values of the evidence objects to make the program more readable. The only relevant evidence objects are ε_2, which supports the judgment $[\mathit{transform} : \mathsf{Int}, \mathit{size} : \mathsf{Int} \to \mathsf{Int}, \cdot] \lesssim ?$, $\mathit{idom}(\varepsilon_1)$, which supports the judgment $? \lesssim ?$, and ε_4, which supports the judgment $? \lesssim [\mathit{size} : \mathsf{Int}, \mathit{transform} : \mathsf{Int} \to \mathsf{Int}, \cdot]$.

This program first reduces using the [Beta-Succ] notion of reduction, a rule that computes $\varepsilon_2; \mathit{idom}(\varepsilon_1) = \varepsilon_9$ and substitutes the identifier **input** by the value $\varepsilon_9\mathbf{u}$, where $\mathbf{u}$ is the value $[\mathit{transform} = 2, \mathit{size} = \lambda\mathbf{x}\,.\,\mathbf{x}]$.

The program then reduces using contextual reduction, computing $\varepsilon_9; \varepsilon_4$ to a failure, and evaluation stops following the rule $\dfrac{\varepsilon_9; \varepsilon_4 = \bot}{E[F[\varepsilon_4\varepsilon_9\mathbf{u}]] \longrightarrow \mathbf{error}}$. The failing evidence composition operation in the premise identifies two separate inconsistencies, one for

$$\frac{\begin{array}{c}\mathsf{fwd}_{\pi_2(\varepsilon_1)\sqcap\pi_1(\varepsilon_2)=S_{\mathrm{mid}}}\ (\pi_2(\mathring{\varepsilon}_1),\pi_1(\mathring{\varepsilon}_2)) = \mathring{S}_{\mathrm{mid}} \\ \mathsf{fwd}_{\mathcal{I}[\![\pi_1(\varepsilon_1)\lesssim S_{\mathrm{mid}}]\!]=\varepsilon_3}\ \left(\pi_1(\mathring{\varepsilon}_1),\mathring{S}_{\mathrm{mid}}\right) = \mathring{\varepsilon}_3 \qquad \mathsf{fwd}_{\mathcal{I}[\![S_{\mathrm{mid}}\lesssim\pi_2(\varepsilon_2)]\!]=\varepsilon_4}\ \left(\mathring{S}_{\mathrm{mid}},\pi_2(\mathring{\varepsilon}_2)\right) = \mathring{\varepsilon}_4 \\ \mathsf{fwd}_{\mathcal{I}[\![\pi_1(\varepsilon_3)\lesssim\pi_2(\varepsilon_4)]\!]=\varepsilon_5}\ (\pi_1(\mathring{\varepsilon}_3),\pi_2(\mathring{\varepsilon}_4)) = \mathring{\varepsilon}_5\end{array}}{\mathsf{fwd}_{\varepsilon_1;\varepsilon_2=\varepsilon_5}\ (\mathring{\varepsilon}_1,\mathring{\varepsilon}_2) = \mathring{\varepsilon}_5}$$

$$\frac{\mathsf{fwd}_{\pi_2(\varepsilon_1)\sqcap\pi_1(\varepsilon_2)=\bot}\ (\pi_2(\mathring{\varepsilon}_1),\pi_1(\mathring{\varepsilon}_2)) = \bot}{\mathsf{fwd}_{\varepsilon_1;\varepsilon_2=\bot}\ (\mathring{\varepsilon}_1,\mathring{\varepsilon}_2) = \bot} \qquad \frac{\mathsf{fwd}_{\pi_2(\varepsilon_1)\sqcap\pi_1(\varepsilon_2)=\bot}\ (\pi_2(\mathring{\varepsilon}_1),\pi_1(\mathring{\varepsilon}_2)) = \blacksquare}{\mathsf{fwd}_{\varepsilon_1;\varepsilon_2=\bot}\ (\mathring{\varepsilon}_1,\mathring{\varepsilon}_2) = \langle\blacksquare,\blacksquare\rangle}$$

$$\frac{\begin{array}{c}\mathsf{fwd}_{\pi_2(\varepsilon_1)\sqcap\pi_1(\varepsilon_2)=S_{\mathrm{mid}}}\ (\pi_2(\mathring{\varepsilon}_1),\pi_1(\mathring{\varepsilon}_2)) = \mathring{S}_{\mathrm{mid}} \\ \mathsf{fwd}_{\mathcal{I}[\![\pi_1(\varepsilon_1)\lesssim S_{\mathrm{mid}}]\!]=\bot}\ \left(\pi_1(\mathring{\varepsilon}_1),\mathring{S}_{\mathrm{mid}}\right) = \varepsilon^{\circ}_{\bot} \qquad \mathsf{fwd}_{\mathcal{I}[\![S_{\mathrm{mid}}\lesssim\pi_2(\varepsilon_2)]\!]=\varepsilon_4}\ \left(\mathring{S}_{\mathrm{mid}},\pi_2(\mathring{\varepsilon}_2)\right) = \mathring{\varepsilon}_4\end{array}}{\mathsf{fwd}_{\varepsilon_1;\varepsilon_2=\bot}\ (\mathring{\varepsilon}_1,\mathring{\varepsilon}_2) = \varepsilon^{\circ}_{\bot}}$$

$$\frac{\begin{array}{c}\mathsf{fwd}_{\pi_2(\varepsilon_1)\sqcap\pi_1(\varepsilon_2)=S_{\mathrm{mid}}}\ (\pi_2(\mathring{\varepsilon}_1),\pi_1(\mathring{\varepsilon}_2)) = \mathring{S}_{\mathrm{mid}} \\ \mathsf{fwd}_{\mathcal{I}[\![\pi_1(\varepsilon_1)\lesssim S_{\mathrm{mid}}]\!]=\varepsilon_3}\ \left(\pi_1(\mathring{\varepsilon}_1),\mathring{S}_{\mathrm{mid}}\right) = \mathring{\varepsilon}_3 \qquad \mathsf{fwd}_{\mathcal{I}[\![S_{\mathrm{mid}}\lesssim\pi_2(\varepsilon_2)]\!]=\bot}\ \left(\mathring{S}_{\mathrm{mid}},\pi_2(\mathring{\varepsilon}_2)\right) = \varepsilon^{\circ}_{\bot}\end{array}}{\mathsf{fwd}_{\varepsilon_1;\varepsilon_2=\bot}\ (\mathring{\varepsilon}_1,\mathring{\varepsilon}_2) = \varepsilon^{\circ}_{\bot}}$$

$$\frac{\begin{array}{c}\mathsf{fwd}_{\pi_2(\varepsilon_1)\sqcap\pi_1(\varepsilon_2)=S_{\mathrm{mid}}}\ (\pi_2(\mathring{\varepsilon}_1),\pi_1(\mathring{\varepsilon}_2)) = \mathring{S}_{\mathrm{mid}} \\ \mathsf{fwd}_{\mathcal{I}[\![\pi_1(\varepsilon_1)\lesssim S_{\mathrm{mid}}]\!]=\bot}\ \left(\pi_1(\mathring{\varepsilon}_1),\mathring{S}_{\mathrm{mid}}\right) = \varepsilon^{\circ}_{\bot 1} \qquad \mathsf{fwd}_{\mathcal{I}[\![S_{\mathrm{mid}}\lesssim\pi_2(\varepsilon_2)]\!]=\bot}\ \left(\mathring{S}_{\mathrm{mid}},\pi_2(\mathring{\varepsilon}_2)\right) = \varepsilon^{\circ}_{\bot 2}\end{array}}{\mathsf{fwd}_{\varepsilon_1;\varepsilon_2=\bot}\ (\mathring{\varepsilon}_1,\mathring{\varepsilon}_2) = \varepsilon^{\circ}_{\bot 1}\sqcap\varepsilon^{\circ}_{\bot 2}}$$

$$\frac{\begin{array}{c}\mathsf{fwd}_{\pi_2(\varepsilon_1)\sqcap\pi_1(\varepsilon_2)=S_{\mathrm{mid}}}\ (\pi_2(\mathring{\varepsilon}_1),\pi_1(\mathring{\varepsilon}_2)) = \mathring{S}_{\mathrm{mid}} \\ \mathsf{fwd}_{\mathcal{I}[\![\pi_1(\varepsilon_1)\lesssim S_{\mathrm{mid}}]\!]=\varepsilon_3}\ \left(\pi_1(\mathring{\varepsilon}_1),\mathring{S}_{\mathrm{mid}}\right) = \mathring{\varepsilon}_3 \qquad \mathsf{fwd}_{\mathcal{I}[\![S_{\mathrm{mid}}\lesssim\pi_2(\varepsilon_2)]\!]=\varepsilon_4}\ \left(\mathring{S}_{\mathrm{mid}},\pi_2(\mathring{\varepsilon}_2)\right) = \mathring{\varepsilon}_4 \\ \mathsf{fwd}_{\mathcal{I}[\![\pi_1(\varepsilon_3)\lesssim\pi_2(\varepsilon_4)]\!]=\bot}\ (\pi_1(\mathring{\varepsilon}_3),\pi_2(\mathring{\varepsilon}_4)) = \varepsilon^{\circ}_{\bot}\end{array}}{\mathsf{fwd}_{\varepsilon_1;\varepsilon_2=\bot}\ (\mathring{\varepsilon}_1,\mathring{\varepsilon}_2) = \varepsilon^{\circ}_{\bot}}$$

Figure 5.12: Forward slicing for evidence composition with BRR

each label in the record types. I can use backward slicing twice to recover the following program slice from the original:

$$\langle \blacksquare \to \blacksquare, \blacksquare \to \blacksquare \rangle\ (\lambda\, \mathbf{input}\,.\,(\blacksquare\blacksquare\ \varepsilon_4\ \mathbf{input}))\ \varepsilon_2\blacksquare$$

Note that the slice of the runtime program only encodes type information in the run-time evidence objects and that the argument that imposes the type constraints (that is, the value of identifier *input*) has been marked as irrelevant. But I still have not backward sliced the elaboration process for this program slice. For exposition purposes, let's focus on backward slicing the elaboration of the outermost function argument, $\varepsilon_2\blacksquare$. The determined backward slicing for an application in BRR is equivalent to the following rule:

$$\frac{\begin{array}{cc} \underset{\Gamma\vdash t_1 \rightsquigarrow \mathbf{e}_1 : S_1}{\mathsf{bwd}} \left(\mathring{\mathbf{e}}_1, \mathring{S}_1\right) = \left(\mathring{\Gamma}, \mathring{t}_1\right) & \underset{\mathcal{I}[\![S_1 \lesssim \widetilde{dom}(S_1) \to \widetilde{cod}(S_1)]\!] = \varepsilon_1}{\mathsf{bwd}} (\mathring{\varepsilon}_1) = \left(\mathring{S}_1, \mathring{S}_3\right) \\ \underset{\Gamma\vdash t_2 \rightsquigarrow \mathbf{e}_2 : S_2}{\mathsf{bwd}} \left(\mathring{\mathbf{e}}_2, \mathring{S}_2\right) = \left(\mathring{\Gamma}, \mathring{t}_2\right) & \underset{\mathcal{I}[\![S_2 \lesssim \widetilde{dom}(S_1)]\!] = \varepsilon_2}{\mathsf{bwd}} (\mathring{\varepsilon}_2) = \left(\mathring{S}_2, \mathring{S}_4\right) \end{array}}{\underset{\Gamma\vdash t_1\ t_2 \rightsquigarrow \varepsilon_1\mathbf{e}_1\ \varepsilon_2\mathbf{e}_2 : \widetilde{cod}(S_1)}{\mathsf{bwd}} \left(\mathring{\varepsilon}_1\mathring{\mathbf{e}}_1\ \mathring{\varepsilon}_2\mathring{\mathbf{e}}_2, \mathring{S}_5\right) = \left(\mathring{\Gamma}, \mathring{t}_1\ \mathring{t}_2\right)}$$

At this point, ε_2 is *very* relevant. Without ε_2 or ε_4, the run-time type inconsistencies never arise. But the original program has no evidence objects in it: it has only types and terms. For backward slicing to obtain $\mathring{t}_2$, it relies on the two last premises of the rule. I begin by computing some gradual type slices $\widetilde{dom}(\mathring{S}_1)$ and $\mathring{S}_2$ using the premise

$$\underset{\mathcal{I}[\![[\mathit{transform}:\mathsf{Int}, \mathit{size}:\mathsf{Int}\to\mathsf{Int}, \cdot] \lesssim ?]\!] = \varepsilon_2}{\mathsf{bwd}} (\varepsilon_2) = ([\mathit{transform} : \mathsf{Int}, \mathit{size} : \mathsf{Int} \to \mathsf{Int}, \cdot], \blacksquare)\,,$$

which produces a relevant gradual type slice for $\mathring{S}_2$. This type constraint and a slice of $\mathbf{e}_2$ (in this case, $\blacksquare$) are the inputs to the computation of

$$\underset{\cdot\vdash[\mathit{transform}=2, \mathit{size}=\lambda(x:\mathsf{Int}).x] \rightsquigarrow \mathbf{u}:[\mathit{transform}:\mathsf{Int}, \mathit{size}:\mathsf{Int}\to\mathsf{Int}, \cdot]}{\mathsf{bwd}} (\blacksquare, [\mathit{transform} : \mathsf{Int}, \mathit{size} : \mathsf{Int} \to \mathsf{Int}, \cdot])\,,$$

which acts on slices of $\mathbf{e}_2$ and $\mathring{S}_2$ to produce a slice of Γ and a slice of t_2. This operation must preserve the structure of the terms that generate the non-irrelevant type constraints, leading to the following slice of the original program:

$$\begin{array}{l} \mathsf{let}\ \mathit{input} : \blacksquare := [\mathit{transform} : 2, \mathit{size} : \lambda(x : \blacksquare).x]\ \mathsf{in} \\ \quad \mathsf{let}\ \mathit{config} : [\mathit{size} : \mathsf{Int}, \mathit{transform} : \blacksquare \to \blacksquare] := \mathit{input}\ \mathsf{in} \\ \qquad \blacksquare \end{array}$$

This slice does not contain only type declarations but also a record value. When non-irrelevant type constraints are implicit, in the sense that they arise from the type of a value, then slicing includes the code that generates the non-irrelevant type constraint.

Although slicing removes parts of the original program irrelevant to the failure, it does not pinpoint a single cause for the failure. Perhaps programmers will see that there is a mismatch between the constraints for *config* and the value of *input*, which is precisely the

problem I injected in the program. Or perhaps programmers will observe a mismatch between the implicit type constraints for *input* and the explicit type declaration for *config*. My hypothesis is that these slices can help programmers identify type mismatches in programs, but this hypothesis cannot be tested with a formal theorem. In Chapter 6, I look for empirical evidence about whether slices are useful for programmers when debugging run-time type errors.

"Pay no attention to that man behind the curtain!"

— The Wizard, in The Wizard of Oz (1939)

6 DEBUGGING RUN-TIME TYPE ERRORS IN PRACTICE

I now present a prototype slicing tool and a user study collecting qualitative feedback from real-world developers. The tool is based on the slicing systems developed in previous chapters. The results from this study provide initial empirical evidence that program slices can be useful for programmers when debugging the run-time type errors that arise in gradual typing.

In previous chapters, I presented design guidelines for formal languages, but focusing on the formal aspects of slicing systems alone cannot provide insight about the usefulness of these systems for developers. To answer questions like "Can developers use a Galois slicing system to identify causes of run-time type failures?", "Will they use it?", "Does such a system help developers?", or "Is the information in the slice consistent with developer expectations?", one must ask developers themselves.

Developers that have experience using gradual languages could provide the most practically meaningful insights about our system. But not many developers use the languages and slicing systems I discuss in Chapter 5. These languages act as proxy models for languages that programmers do use. Although these languages include features present in common programming languages, they are designed so that formalisms are easier to pin down. Attempting to answer these developer-related questions directly in these languages cannot be guaranteed to transfer to real-world programmers and their programming practices. Similarly, making developers use a new tool and at the same time a new, unfamiliar, and limited language could distract participants from the tool and bias the results of the study. At the same time, building a slicing system for a real-world programming language requires addressing the many particularities and corner cases in a language implementation. Real-world programming languages are complex and giant systems, lacking the semantic clarity and complete specification of formalisms. These considerable engineering efforts are beyond the scope of the research project for this dissertation.

I take a middle ground between these two options by developing an experiment using the "Wizard of Oz" (Kelley, 1983) methodology. In a Wizard of Oz study, the experimenter acts as a puppeteer[2] and simulates the system to be tested, allowing participants to use the system and provide feedback without requiring a complete implementation. This type of experiment is intrinsically limited as it does not allow participants to freely alter the programs they debug or try their own programs. Still, it allows me to evaluate program slicing in the context of a general-purpose language with a large participant pool of real-world programmers.

The experiment I designed presents participants with a programming interface extended with mockup slices and a system to present them, for a predetermined set of

example programs in a general-purpose programming language. To ensure the slices presented to programmers are correct, I rely on the semantics of GTFL$_{\lesssim}$ described in previous chapters to obtain program slices for semantically equivalent programs for each task, and I manually adapt the syntax from GTFL$_{\lesssim}$ to the general-purpose programming language.

I choose TypeScript as the general-purpose programming language for the study. As discussed in Chapter 2, not many real-world languages called "gradual" are fully gradual, because they do not check for consistency of the type constraints as the program runs. TypeScript is one such language as its optional type annotations do not introduce any extra type checks when the program runs. Still, TypeScript programmers abound. I motivated the tool to participants as a type-safe runtime semantics[3] that detects run-time type inconsistencies. To debug these inconsistencies, I propose and evaluate program slices.

My hypothesis is that the information presented by the tool can help programmers focus their attention on the most relevant portions of the code and make it easier to identify parts of the program that must be changed to resolve the type inconsistency. Although the experiment evaluates slices in a real-world context, it is intended to provide only exploratory and qualitative data about the experiences of some programmers with the tool and collect evidence on whether those programmers find the tool useful. The rest of this chapter first introduces my research question and the design of the experiment, I then discuss the programs in the experiment and comment on the study's results and threats to validity.

6.1 THE WIZARD-OF-OZ USER INTERFACE

Participants of the study are presented with a programming interface that, when a program runs to a type inconsistency, automatically blurs out the portions of the program that a slice guarantees irrelevant to the failure. I developed this online tool based upon an open source web text editor, Microsoft's Monaco.[4] This editor includes TypeScript syntax highlighting and integrates a TypeScript compiler providing access to type checking, compiling, and type information from the web browser. I extended the base text editor to include:

- The option for participants to highlight multiple parts of the program. These highlights can also be removed.
- A way to present a program slice as a layer on top of the code. Parts of the original program marked in the slice as "irrelevant" appear in the interface as blurred text. I chose blurring to drive attention away from the irrelevant parts instead of towards them. I did not choose highlighting to avoid confusion between the slice and participant highlighting. I did not choose underlining to avoid confusion with the underlining already provided by the editor when a program fails to compile. When a slice is available, the text editor also allows for the test subjects to hide the slice and see the full program text.
- A "console output" viewer that presents some text when participants click on the "Run" button.

Due to limitations of the Wizard-of-Oz approach, participants were not allowed to change the program code of the program. This restriction also ensured that participant highlights remained comparable. Participants could still access the feedback tools from the Monaco editor, as visible in Figure 6.2: hovering over an identifier shows the type TypeScript assigns to it, and clicking on an identifier highlights all other references to the same identifier in the program text.

The program slice and the console output constitute the Wizard-of-Oz portion of the experiment and are provided by the experimenter, not directly computed by a tool in the experiment. The slices are instead computed by representing the programs as closely as possible in the $\text{GTFL}_{\lesssim}$ language and using an interpreter for the slicing semantics discussed in previous chapters to obtain a corresponding slice.

6.2 RESEARCH QUESTIONS

The goal of this user study is to validate whether programmers can use program slices to debug run-time type errors. But this question can be answered only indirectly, as programmers are exposed to a particular implementation of a tool that presents them with a slice. I focus on three related research questions: **(Q1)** *Can developers use the slicing interface I designed to identify causes of run-time type failures and to justify changes to the program that avoid the previous failures?* This question provides some empirical support to the hypothesis that developers will use slices when available and that the information in the slices is helpful for debugging. A core contribution of my dissertation is to apply slices to the domain of run-time type failures. I intend slices to be used as debugging aids, so developers can identify causes of failures and propose fixes or changes to the code. If developers cannot use slices to achieve this goal, then the theoretical contributions of slices are less relevant.

This question is related to other secondary questions: **(Q2)** *Do developers find slices useful?* , and **(Q3)** *Do developers rely on slice information to fix program errors?* Both questions hint at whether developers would trust and use slices. If developers perceive slices as useful and rely on their information, they are more likely to use a system in the future.

6.3 EXPERIMENT DESIGN

I propose a formative randomized user study to explore the questions in Section 6.2. I ran the study online in a remote fashion, using the Prolific platform (`https://prolific.com`) as a recruiting tool for participants in the study. This online platform acts as an intermediary for research studies, managing payments and providing limited ways to screen for participants.

Participants were first invited to answer a pre-screening questionnaire. Those successful in the pre-screening were invited to the main study, consisting of three parts: a tutorial, a set of tasks, and an exit questionnaire.

6.3.1 *Pre-screening*

Participants needed to have worked in the Software industry and have familiarity with TypeScript programming to be included in this experiment. Only the first criterion was screened by the Prolific platform itself, so I required candidates to answer a pre-screening questionnaire to further refine the participant pool to those familiar with TypeScript. All participants who completed the consent form were paid £1.20 immediately after finishing the questionnaire, even if the participant was screened out. I intended the questionnaire to take no longer than 10 minutes, although this limit was not enforced. Participants that completed the questionnaire appropriately were then invited to participate in the main study. Participants that completed the main study were paid £11.00.

This questionnaire checks whether participants are familiar with the language of interest (TypeScript) and understand type constraints in TypeScript programs. Participants were not told directly about the participation criteria. Instead, I asked generally what languages the candidate was familiar with. Candidates that mentioned familiarity with TypeScript were then provided with four short program options and asked to select the ones that were written in TypeScript (only one was). Programmers that succeeded in detecting the TypeScript program and rejecting the others were then provided with a second TypeScript program and a list of statements about the type constraints in that program and were asked to select all the correct statements.

Because the questionnaire did not provide access to any tool support usually available to TypeScript developers, many participants were unable to completely detect all type constraints or were confused by subtleties in the semantics of TypeScript, as I discuss in the results section.

6.3.2 *Main study setup*

The main study consists of three parts: a tutorial, a set of six tasks, and an exit questionnaire. Candidates that succeeded on the pre-screening were invited to participate in the main study. Not every successful candidate participated on the final study, and several participants "returned" their study submissions, an option provided by Prolific to escape participation without payment and equivalent to withdrawing consent.

6.3.2.1 *Tutorial*

Participants invited to the main study completed a brief, fully guided tutorial task (10 minutes) to ensure that they meet the technical requirements for the main study. The tutorial introduced the tool and guided participants through highlighting tasks just like the ones they later faced in the experiment. It also served as an opportunity to remind participants of the lack of type safety in TypeScript, with a program where type inconsistencies are not detected by the compiler and lead the program to fail.

During the tutorial, participants are made aware of the type safety limitations of the TypeScript language, which motivates "TypeSlicer", a fantasy name for a Galois slicing system. The tool is described to participants as providing "*two features:*

1. *It tracks type information and checks for inconsistencies throughout program evaluation.*

```
var person: {id:string} = {id:'hello'}
var base_id: number = 1
function add_id (x: {}) : {id? : number } {
  let ans : {id? : number} = x
  ans.id = base_id++
  return ans
}

var z = add_id(person)

console.log(person.id.toUpperCase())
```

Figure 6.1: Tutorial program, highlighting the lack of type safety in TypeScript

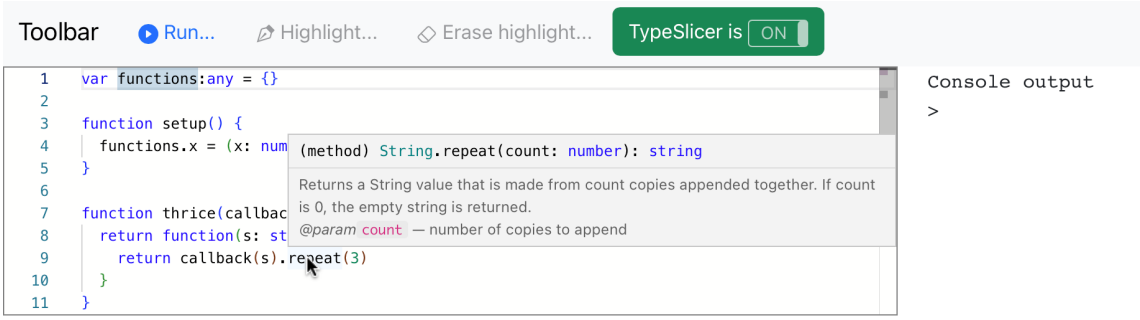


Figure 6.2: Text editor interface used in the main study.

2. *Once it detects a type contradiction, TypeSlicer backtracks evaluation and identifies the parts of the program that were not involved in the contradiction.*

The user interface is then able to hide parts of the code guaranteed not to be involved in the contradiction."

The program in Figure 6.1 was presented as part of the tutorial. The TypeScript compiler accepts this program with no warnings, but it runs to a run-time type error in the last line, as after calling the function `add_id`, the object `person` contains a number in field `id` instead of a string. This is one of the (many) situations where TypeScript is not type safe.[5]

6.3.2.2 *Debugging tasks*

After the tutorial, all participants were presented with 6 debugging tasks. In each task, participants were presented with a text editor containing a program to debug in the interface presented in Figure 6.2. The program compiles successfully but runs to an error. A console window to the right presents error output when the program is "Run" by clicking a button. Clicking the button only simulates the run, providing instead a stored error message and a stored program slice. The toolbar provides access to the program running simulation, a green toggle for the TypeSlicer tool when available, and highlighting facilities after the program is "run". The task description lists three actions to perform:

1. Run the code, and think of candidate fixes for this error, and highlight the parts of the program that need to be changed to fix the bug. All highlights were recorded.

2. To clarify why participants highlighted specific program fragments, the task description asks participants to report their understanding of the failure by writing a "change recommendation" that would be attached to a bug report of the failure.

3. Write any comments on the program behaviour and whether anything was surprising or unexpected.

Participants did not need to provide new code, but were asked to describe a possible solution to the programming bugs presented, reflecting their understanding of the program.

Each debugging task was expected to be approximately 5 to 10 minutes long, with 15 minutes being the maximum allowed time before participants were required to move on to the next task. All participants were shown the same set of six programs, but to account for fatigue and learning effects, the order in which the programs were shown was randomized. Participants had access to the tool in 3 of the programs and the other 3 acted as a control setting, so participants did not have access to the tool. The choice of programs where the tool was available was also randomized.

6.3.2.3 *Exit questionnaire*

The third part of the study was an exit questionnaire where participants were free to provide comments about the tool, the programs, or the errors they were faced with. Finally, they were asked to answer a standard usability survey (the System Usability Scale of Brooke (1996)).

The survey intended to measure whether usability issues with the tool itself hampered participants (as opposed to the information available from the slices). This usability scale is a popular "cheap to apply" process that can act as a baseline against which to measure bad usability of an interface (Brooke, 2013). System Usability Scale scores provide additional feedback on user perceptions about the interface I designed..

6.4 SELECTION OF PROGRAMS FOR USER STUDY

I selected 6 programs for this study, representing run-time type errors with different levels of complexity. I both designed small problems for the study and adapted some example programs from related literature on blame (Lazarek, King, et al. (2019)) and gradual typing evaluation (Tunnell Wilson et al. (2018)). I made each program self contained and a 5 to 10 minute debugging session a realistic goal for each program.[6] The rest of this section describes the programs and the bugs that they have. Sections highlighted in grey represent the portions of the program that would be blurred out by TypeSlicer during the study.

6.4.1 *Program 1: Wrong-Map*

In Program 1, a call to the `example` function relies on a ill-implemented `map` function. The `map` function is wrong, because it applies its functional argument `f` only to the first element of the Array. The type signature for `map` is weak and is never broken, but the use

of map assumes it completely transforms its input into an array of numbers, the type of the range of `f`. But `map` returns an array that can contain either numbers or strings instead of being constrained to contain only numbers.

```
function map ( f: (x: any) => any , l: Array< any >): Array< any > {
  if (l.length === 0) {
    return []
  }
  return [ f(l[0]) ].concat(l.slice(1))
}

function example (list : Array<String>) : Array<Number> {
  return map( (x: String) => { return x.length } , list)
}

console.assert (
  example([ "a" , "b" , "c" ]).every((x: Number) => x === 1 ),
  "example must produce a list of 1's"
)
```

This program was included because the type inconsistency arises in the body of `example`, although the implementation issue is in `map`. This program is a simplification of an example problem described in the analysis of blame error reports by Lazarek, King, et al. (2019), a mutation of a program implementing an infinite stream of prime numbers (Sieve). I expect slices to draw the attention of participants to part of the implementation of `map` instead of focusing on the implementation of `example` only.

6.4.2 *Program 2 : Wrap*

In this program, the `wrap_again` function does not statically check the type of its argument, because any function parameter without a type declaration defaults to `any` in TypeScript. The call to `wrap_again` on line 1 manages to hide the type of a function that returns a string and present it to the rest of the program as a function returning a number that is stored in the `container` variable.[7] The problem arises when multiplying the result of `f()` on line 12, but the problem is far removed in a set of stacked type wrappers.

This program is a simplification of an example program identified by Lazarek, King, et al. (2019) (Fig. 13, a mutation of KCFA), which they find interesting for the evaluation of blame tracking with contracts.[8] Although the original problem does not arise in AGT languages, I keep the program as a set of nested type constraints where the path of inconsistency is somewhat complex. I expect slices to draw participants' attention away from the implementation of `main` and towards the argument to `container`.

```
1  let container = wrap_again( () => "2000")
2
3  function wrap (x: () => number): () => number {
4    return x
5  }
6
7  function wrap_again (x) : () => number {
8    return wrap(x)
```

```
9  }
10
11  function main (f: () => number): bigint {
12    return BigInt( f()* 10000 )
13  }
14
15 console.assert(
16    19999999<= main(container) ,
17    "result is big enough"
18  )
```

6.4.3 *Program 3 : 3-Strings*

In this program, an incorrect bound in the `for` loop in method `asks_for_2_strings` extracts 3 strings instead of 2. This is not an immediate issue, because the list actually has 3 strings at the beginning but forces the next method call (`asks_for_1_string`) to fail even though the method itself has no problems.

```
1  let hide = function(x) { return x }
2
3  let strings = hide([ "-1" , "0" , "1" , 2, "3", "4", "5" ])
4
5  function next_string() {
6    return strings.pop()
7  }
8
9  function strings_only(x: string): void {
10  console.log( x.toUpperCase() )
11 }
12
13 function asks_for_2_strings(){
14   for (var i = 0; i <= 2; ++i) {
15     strings_only( next_string() )
16   }
17 }
18
19 function asks_for_1_string(){
20   strings_only( next_string() )
21 }
22
23 asks_for_2_strings()
24 asks_for_1_string()
```

This program reflects a failure caused by a problem that is not in the immediate evaluation stack. I simplified a program by Lazarek, King, et al. (2019) (Fig. 12, a mutation of Dungeon). This program exemplifies a situation where the expectations from using a "blame shifting" debugging approach do not hold. The identified component was not the cause of the failure, and refining the type constraints of `asks_for_1_string` does not change the error allocation. This breaks the "blame trail" property of blame shifting.

I include this program to evaluate whether slices prove helpful in a situation where comparable debugging approaches are known to be insufficient.

6.4.4 *Program 4: Non-uniform list*

This simple program is a variation on "Program 2" proposed by Tunnell Wilson et al. (2018). The list `t` is not uniform, and the element selected of the list, `strs[1]`, is a number and not a string. Although the first three lines should suffice to make an error in a type safe language, they do not suffice in TypeScript. The fourth line was added to guarantee that the program produces an error when the type safety checking tool is not in use.

```
1 let t: any = [ "A", 3]
2 let strs: Array<string> = t
3 let fst: string = strs[1]
4 fst.toUpperCase()
```

This is a simple example without any branching and where the type annotations do not correspond to the values hiding behind a list of type `any`. Tunnell Wilson et al. (2018) identified that some developers expected the program to fail on line 2, where the non-uniform array was cast to an array of strings instead of when the array is accessed. This suggests *"that developers prefer a strategy that type-checks the elements of an untyped array immediately when the array flows into a typed context instead of waiting until the array is accessed."* (Tunnell Wilson et al., 2018). I expect this short example to provide an easier example through the experiment and that the highlights will vary among participants on where to change the program.

6.4.5 *Program 5: Push element*

The following two programs are new. Like in program 4, program 5 presents a single type inconsistency crossing through an `any` type annotation, but it has a slightly more complex control flow than program 4. I introduce this program as a simple example where the inconsistency is not immediately apparent when reading the program top to bottom.

The `any` crossing happens at the parameter of the `process` function. The parameter is an array of strings, but the array is then used in the function at an inconsistent type by pushing a new value into it that is a number (`y.length`).

```
function process(y): void {
  let z: Array<number> = y
  z.unshift(y.length)
}

let x: Array<string> = ["hello", "bye"]
process(x)
console.assert(
  x[0] === "2",
  "List includes its length at the head"
)
```

6.4.6 *Program 6: Thrice*

In this program, I introduce a more complex inconsistency where the failure location misleads from the original incorrect location. I expect slices to guide participants towards the problem source in this example.

This program uses higher-order types repeatedly as callbacks. The function `thrice` produces a function that applies a stored callback to produce a string, which is then repeated three times. The function `thrice` itself is correct, but its use in line 16 (in the body of `action`) passes a function of the wrong type (number to number) that the type checker can only see as `any`.

```
1  var functions:any = {}
2
3  function setup() {
4    functions.x = (x: number) => Math.pow(x,2)
5  }
6
7  function thrice(callback: (x: string) => string) : (s: string) => string {
8    return function(s: string): string {
9      return callback(s).repeat(3)
10   }
11 }
12
13 setup()
14
15 function action( y: string )  : string  {
16     return  thrice(functions.x)( y )
17 }
18
19 console.assert(
20   action( "hello" )  === "25 25 25 ",
21    "Thrice the length squared"
22 )
```

6.5 RESULTS

I describe the data I obtained from the study and propose some hypotheses and future research directions.

6.5.1 *Participant population*

The pre-screening survey filtered candidate participants for the study. Out of Prolific's pool of 2,215 participants with Software industry experience, 259 participants filled the survey. 60 answered that they had written at least 1000 LOC in TypeScript, but eight of them could not clearly identify the only valid TypeScript Program among four options.

I presented the other 52 candidates with an opportunity to identify the type constraints in a program. Only 11 provided the completely correct answer. Parts of the answer

highlighted subtleties with TypeScript types that went undetected by many participants when presented with a plain text program but which would have been visible in any development environment.

Most subtleties arose on the treatment of `any` as a type. First, when local variables in a TypeScript program are not annotated with a type, the type checker infers their type. This is not the case for unannotated function parameters, which default to type `any`. Programmers may forget this distinction and expect the type of function parameters to also be inferred to a more precise type. Second, any expression can be assigned a type `any`, even when the type system can infer a more precise type. Programmers may then expect statements of the form "X has type any" to always hold, even when the compiler can infer a more precise type for X.

Another subtlety arose with object types. One of the type annotations in the survey gave the identifier `person` the type `{ x:string }`, representing an object with a single field `x` of type string. 17 people marked the statement "`person` has type `string`" as correct, which is false. Participants might have not paid close attention to the subtle difference between `{ x:string }` and `string`. However, any attempt to use the object as a string would have produced a compiler type error requiring only a minor correction.

I opened the experiment first to participants who answered all types correctly, obtaining 10 submissions. I then opened the experiment to participants whose responses had issues with subtleties of `any`, and obtained 14 more responses. I then opened the experiment to participants whose responses had the incorrect type for person, receiving 5 more responses for a total of 29.

For the main study, I collected a total of 174 individual tasks from the 29 participants that finished the study. The sample size of the study is close to the average sample size found in HCI user studies (Caine, 2016).[9] Participants exceeded the 15-minute timeout in only 11 of the 174 tasks (6%).

6.5.2 *Slices affect developer focus*

The main portion of the study consisted of participants highlighting "every part of the code that needs to be changed" and providing a change recommendation. I aggregated all the highlights for each program as a heat map in Figures 6.3 to 6.8, where the highlighting intensity correlates with how many participants highlighted that section of the program. In each figure, I separately show the highlights made by participants that had the slicing tool available from those who did not.

In some cases (Programs 1, 2 and 4, in Figures 6.3 to 6.5), highlights tended to concentrate in fewer parts of the program when the tool was available, suggesting that slices can help in narrowing the places where programmers pay attention during debugging. However, this concentration did not always happen. Notable counterexamples are Figures 6.6 and 6.7, which are interesting as they show clear differences on the program parts most highlighted by participants.

I discuss program 3 from Figure 6.6 first. When the slicing tool was not available, most programmers picked line 10 as an issue. When the slicing tool was available, more participants were inclined to change the contents of the `strings` array on line 3. This change of focus suggests that slices helped participants understand the programs, and

```
function map (f: (x: any) => any, l: Array<any>): Array<any> {
  if (l.length === 0) {
    return []
  }
  return [f(l[0])].concat(l.slice(1))
}

function example (list : Array<String>) : Array<Number> {
  return map((x: String) => { return x.length }, list)
}

console.assert(
  example(["a", "b", "c"]).every((x: Number) => x === 1),
  "example must produce a list of 1's"
)
```

Without slices (n = 19)

```
function map (f: (x: any) => any, l: Array<any>): Array<any> {
  if (l.length === 0) {
    return []
  }
  return [f(l[0])].concat(l.slice(1))
}

function example (list : Array<String>) : Array<Number> {
  return map((x: String) => { return x.length }, list)
}

console.assert(
  example(["a", "b", "c"]).every((x: Number) => x === 1),
  "example must produce a list of 1's"
)
```

With slices (n = 10)

Figure 6.3: Heat maps of participant highlights - Program 1

```
let container = wrap_again( () => "2000")

function wrap (x: () => number): () => number {
  return x
}

function wrap_again (x) : () => number {
  return wrap(x)
}

function main (f: () => number): bigint {
  return BigInt( f()*10000 )
}

console.assert(
  19999999<= main(container),
  "result is big enough"
)
```

Without slices (n = 11)

```
let container = wrap_again( () => "2000")

function wrap (x: () => number): () => number {
  return x
}

function wrap_again (x) : () => number {
  return wrap(x)
}

function main (f: () => number): bigint {
  return BigInt( f()*10000 )
}

console.assert(
  19999999<= main(container),
  "result is big enough"
)
```

With slices (n = 18)

Figure 6.4: Heat maps of participant highlights - Program 2

```
let t: any = ["A", 3]
let strs: Array<string> = t
let fst: string = strs[1]
fst.toUpperCase()
```

Without slices (n = 13)

```
let t: any = ["A", 3]
let strs: Array<string> = t
let fst: string = strs[1]
fst.toUpperCase()
```

With slices (n = 16)

Figure 6.5: Heat maps of participant highlights - Program 4

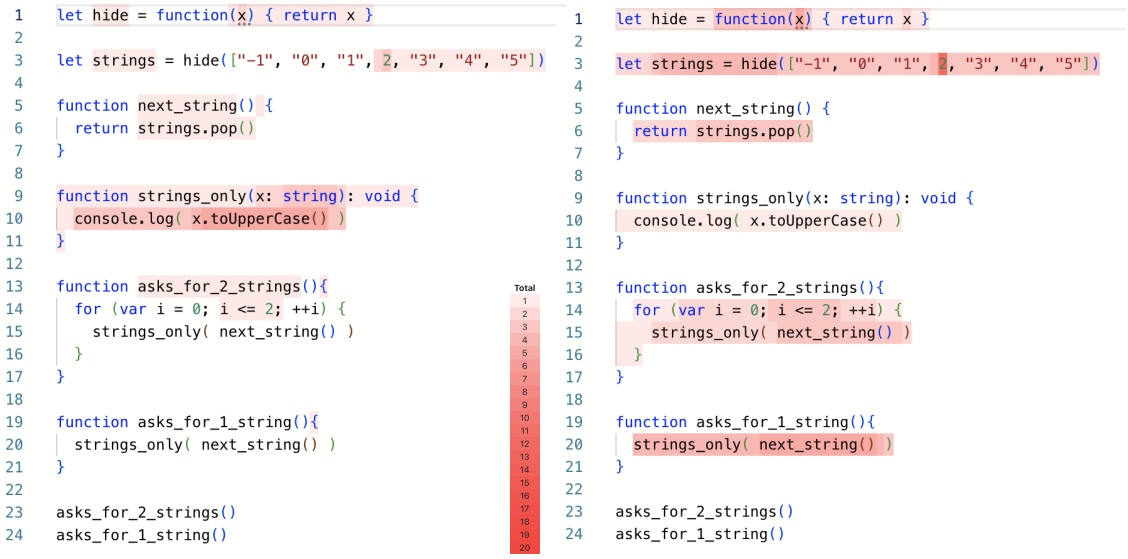


Figure 6.6: Heat maps of participant highlights - Program 3

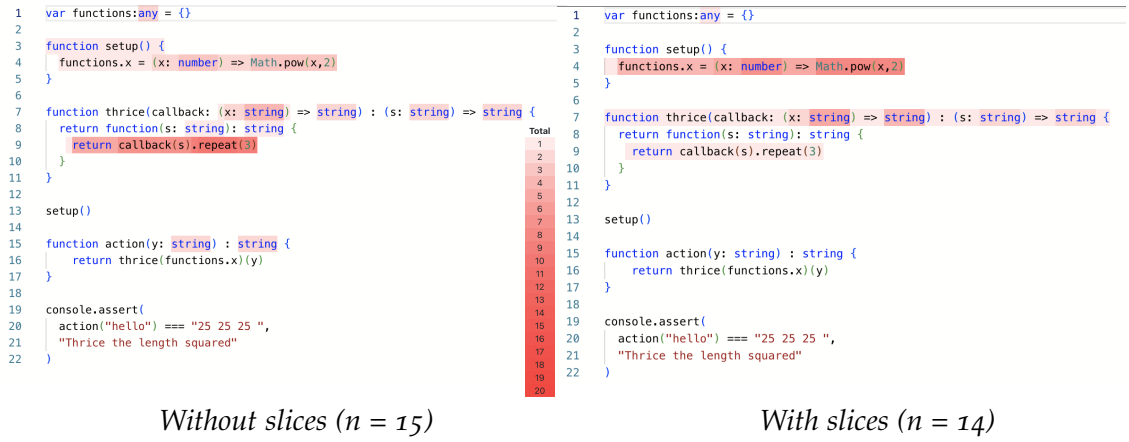
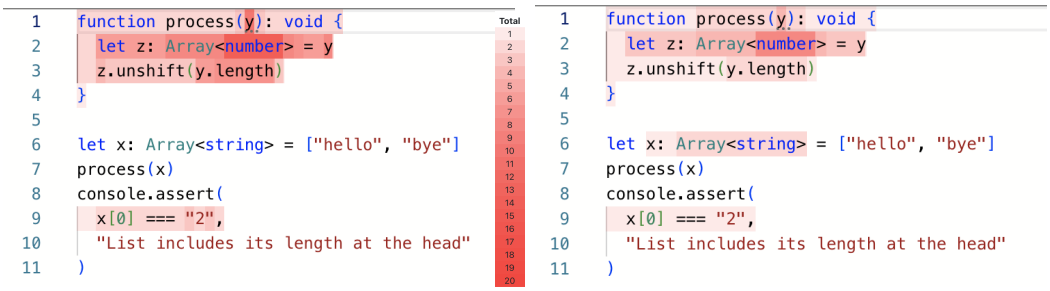


Figure 6.7: Heat maps of participant highlights - Program 6

directed what they considered correct and incorrect: no change is required on line 10 if one considers types as correct, while the changes on line 3 are more consistent with considering the type annotation for `strings_only` as correct.

Program 6 shows similar results. Although the implementation of `thrice` is consistent with its type annotation, most participants without access to the tool proposed changing the implementation of `thrice`, which is where the bug would immediately manifest. On the other hand, most participants with access to the tool proposed changing the implementation of `functions.x`, the first argument given to `thrice`, which does not have the type signature expected for the argument of `thrice`.

The change recommendations provided by participants support this distinction. Without the tool, many participants assumed that functions.x was correct, so they suggested changes to the implementation (and signature) of `thrice` to deal with numbers (changes like "`* 3`" instead of "`.repeat(3)`", comments like "repeat doesn't work with numbers,



Without slices (n = 18) *With slices (n = 11)*

Figure 6.8: Heat maps of participant highlights - Program 5

only strings"), even when there were no type constraints on the functions object (it has type `any`) and both the assertion and most other type declarations mentioned strings. This is an interesting result on its own, as it highlights that at least in some situations, programmers do side with the implementation details in spite of type annotations and may assume the annotations in the program to be incorrect. With the tool, most people suggesting changing `functions.x` to produce a string instead.

From the highlighting data, I make the following qualitative observations:

- Programmers can use the slicing interface, and the information in the slice can alter the overall areas of focus for developers debugging a run-time type error **(Q1)**.
- Slices have an impact on programmer's perceptions of what can be changed in a program. This provides some evidence that developers relied on the slicing tool **(Q3)**.

The data also support observations not directly linked to the research questions but relevant to my dissertation. First, programmers have varied perceptions of what needs to be changed in a program from a run-time type error. Second, programmers recognize different ways to fix a run-time type inconsistency: there is no single solution for program bugs. Finally, programmers do not always consider types as correct, and they tended to propose fewer changes to types when the tool was available in this study.

6.5.2.1 *Participant feedback*

Participants responded positively to the tool. Participants graded the tool at the end of the experiment with an average of 75.7 ($\pm$14.4) points on the System Usability Scale. These scores are out of 100, but they are best considered in comparison with other results. For comparison, Bangor, Kortum, and Miller (2008) report an average score of 70.14 over 2,324 surveys collected in the span of a decade. According to Bangor, Kortum, and Miller (2008), our average score can be considered *"Good"*, with the caveat that *"scores in the 70s and 80s, although promising, do not guarantee high acceptability in the field"*. Although these scores are not perfect, they are sufficient for the purpose of the study: the tool itself is not unacceptably bad as to derail the rest of my investigation.

This overall sentiment is supported by participant comments as well. One participant commented that *"Typeslicer helpe[d] see where the problem is. I liked how it stopped half way*

through the array to see where the problem happened at the int.". Another commented that *"it was quite helpful as in task number 6 [(problem 2)] it showed me only the lines which should be corrected and after checking the code, I saw it made a good decision"*, and another that *"I had really big issues with finding the error without the Slicer. But, with the slicer it was 100x faster, the way it works, blurring out what it's not important really helps to focus on the problem and try to find the solution."*

Participants explicitly mentioned some of the programs given in the experiment as being hard to understand or debug. I selected more complex programs for this experiment than other related studies (Tunnell Wilson et al., 2018), so these answers were not surprising. Interestingly, the presence of the tool seems to have made a difference in perceived task complexity. Some participants with access to the slicing tool mentioned that program 6 *"seems unnecessarily complex"* or *"Kind of verbose"*, but in the individual tasks where the tool was not available, the comments turned for the worse: comments like *"Too convoluted"*, or *"Honestly I found this very challenging"*.

Participants expressed a desire for access to the tool when the tool was missing from a particular task: one participant explicitly mentioned twice that the task was harder and type slicer would help. Another participant commented in program 1, that *"it took slightly longer without the TypeSlicer tool"*. This feedback suggests that a slicing tool is helpful in debugging more complex type inconsistencies, but of course more data would be necessary to support this hypothesis.

Although participants were invited to provide comments as part of each task, most participants provided very few comments on individual tasks aside from the change recommendation. In general, no participant mentioned anything surprising about the tool output for any task, only commenting about the program behaviour and the interface. Still, general awareness that the study was looking for open feedback helped generate the extensive positive feedback received in the exit survey and discussed in the next subsection.

6.5.3 *From the slice to the code and back*

Do the test subjects make use of the slicing information, or do they ignore the slice interface? I look for hints of an answer to question **(Q2)** with the data in Figure 6.9. The histogram shows how many times participants toggled the slice information off or on per task when the tool was available. Because when the tool is available for a task it provides its feedback automatically, zero clicks on the histogram means participants only saw the program slice and never looked at the complete program in a particular task. In almost a third of the task runs where the slicing tool was available (31.03% out of 87 tasks), participants never toggled the slice information, basing their participation only on the tool outcome. This suggests that the slicing information was sufficient for them. In 20.69% of the total tasks, the participants at some point turned the slice off and did not reactivate it. About half of those (11.49% of the total tasks) turned the tool off in the first 15 seconds after running the code, which may indicate cases where the participants were not interested in the tool results. At the same time many participants went back and forth between the slice and the code: the toggle average per task was 3.45 ($\pm$4.67).

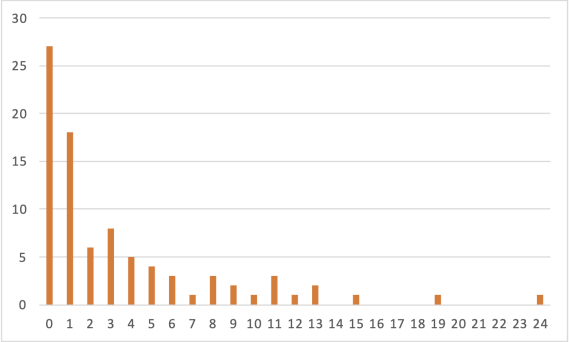


Figure 6.9: Histogram of total toggles of the TypeSlicer tool per individual task

The histogram suggests participants followed varied approaches when using the tool. Lack of toggling might also signal that some participants did not realize that they could toggle the tool or that the tool always provided sufficient information. On the other extreme, repeated toggling might be just a consequence of exploring the interface (one participant commented that *"It's an interesting tool to use - I kept toggling it on and off when it was available so it was a bit distracting"*) . But because the tool blurs the code not on the slice, toggling the tool could also be considered as a proxy for switching attention between the slice and the rest of the program or as a way for programmers to look for extra context on programs they were unfamiliar with.

Overall feedback about the tool was positive and seems to indicate that participants found slices to be useful. Participant comments included

- *"The TypeSlicer helped me in getting clues about where the bugs where in the programs I was debugging. It is a very nifty tool that I would like to use."*
- *"TypeSlicer has definitely helped me pay attention to where and why errors occur in code."*
- *"I think it's a very useful tool and can improve the productivity when working with TypeScript projects"*
- *"I found it to be helpful to some degree and the sort of thing I would use in real life"*

User feedback seems to indicate that in some situations, users prefer accessing the slice and the code together, which the tool did not accommodate directly: negative comments suggested that the tool would be improved by highlighting or underlining instead of blurring the code. Some of the negative feedback included comments like *"It was decent, but sometimes blurring the code just made it harder to debug."* and *"I don't think I would use it. I mean, it could be pretty useful if it would markup code instead of blurring the rest."*. Still, one participant mentioned that *"By blurring individual pieces of code, TypeSlicer definitely made debugging easier"*.

Further study and refinement of the tool is needed to make any conclusions, including checking for visual momentum (Alwis and Murphy, 2006; Woods, 1984) as a measure for disorientation when using the tool, and whether familiarity with the tool and the code base leads developers to focus on the highlighted slice only.

6.6 threats to validity

Some limitations and threats to the results I present in this chapter include:

- **The study evaluates the tool instead of slices themselves.** This experiment can only evaluate whether programmers use slices through a particular implementation and tool design. I limited the impact of this threat using the System Usability Scale evaluation. If user feedback was consistently negative and the tool also received consistently bad scores in usability, the results would indicate problems with the particular interface proposed. Future research would need to explore whether a different way of presenting slices could still be useful. The experiment found positive feedback for the tool, and as the tool depends on slices to work, it provides anecdotal support for the use of slices.

- **The selection of programs may not generalize to real-world debugging situations.** Besides the difficulty in providing a considerable amount of real-world examples in a single session, I framed the study as an exploratory and qualitative experiment. I focused on providing multiple programs that could be addressed in short sessions with little extra context and little familiarity. I attempted to make the programming environment close to a real world system by reusing off-the-shelf tools. I chose a popular programming language but also attempted to avoid depending on language quirks and advanced typing features in the choice of program examples. The interactions between slices, language quirks, and advanced features should be further studied.

  The Wizard-of-Oz approach provides anecdotal evidence on the initial perceptions and reactions to the tool by users, providing evidence that could be later tested in a larger study. Properly answering these questions requires long term observation and follow ups with developers as they use the tool in their daily practice, beyond the scope of this research project. Future research should also explore whether developers would make regular use of the tool and incorporate it to their programming flow. Observing developers in real-world settings would be beneficial, as familiarity with the tool and the code can be taken into account in a more longitudinal study.

- **The error situations in programs may be rare.** I chose programs and bugs where slicing was potentially useful, which may represent only a fraction of the total errors and bugs in gradual programs. Even when slicing appeared helpful, future research should explore how often the use cases for slicing arise in real world codebases. The supportive feedback from study participants makes me optimistic that the tool would appear useful beyond the limited programs and errors situations presented in the experiment tasks.

- **Programs may be too small to expose limitations of slices.** Before checking if slices are always useful, it was worth checking if they were useful at all. The experimental setup was limited to a single session where I intended to test multiple programs, so "bigger programs" presented a time and attention trade-off against "many programs". Further research is required to evaluate whether slices scale to bigger programs and if there are empirical bounds to their practicality.

- **The overwhelming positive feedback may be a consequence of participant bias.** Dell et al. (2012) report that when participants perceive the study examiners to be also the authors of the tool being studied, they tend to provide more positive feedback. I attempted to limit participant bias by designing the experiment to be performed online, without direct interaction with the tool designers, and by wording the experiment instructions carefully to limit the chances of assuming a correlation between study designers and tool designers. All of these precautions can still be unsuccessful, and participant bias remains a risk that must be considered when evaluating the qualitative feedback I have collected.

- **Lack of a comparison with other state-of-the-art error approaches (e.g. blame).** This is a considerable limitation of the study, but validation of the usefulness of slices *per se* is a first step towards evaluation and comparison with other error reporting approaches in the presence of run-time type errors. I am very interested in comparing slices and blame reports in the future.

6.7 conclusion

Participants in the study can use slices when debugging run-time type errors. I designed a Wizard-of-Oz experiment with a program slicing tool that was sufficiently usable for participants to find the slices useful and to express a preference for having slicing information available.

The sample size and the qualitative setup are not sufficient to apply my observations to broader groups of programmers, and more research is required to generalize results, to improve tool usability, and to explore hypotheses from study observations. Still, the study provides initial empirical evidence that slices are useful, beyond purely theoretical claims.

6.8 NOTES

1. As discussed in Chapter 4, sometimes slicing cannot be exact. In that case, a choice needs to be made on whether slicing will under-approximate or over-approximate. Under-approximating slicing will produce information that is "necessary" to reproduce the failure but include some false negatives: some relevant information could have been marked as "irrelevant". Over-approximating slicing will produce information that is "sufficient" to reproduce the failure but include some false positives: some irrelevant information may not be marked as irrelevant. We choose the latter approach, so that we can trust that anything marked as irrelevant is truly irrelevant to reproduce the failure.

2. The role of the experimenter is akin to the role of the Wizard in the 1939 film "The Wizard of Oz", thus the name.

3. The idea of a type-safe runtime semantics for TypeScript is not novel, and I do not pretend to be introducing it. The semantics of TypeScript is not the focus of my work. Others have built on the work of Rastogi et al. (2015), who proposed a type safe version of the language. However, the cost trade-off has not yet convinced the language designers to adopt a type safe semantics in the official TypeScript language, which remains unsafe.

4. https://microsoft.github.io/monaco-editor/. This editor powers Visual Studio Code, a currently popular text editor and developer environment.

5. Note that this particular kind of run-time type error is not caused by gradual type annotations, but instead by interactions between mutation and record subtyping: the body of `add_id` assumes that it is safe to extend its argument with new fields, as its type declares it to be an empty object (or, equivalently for purposes of the example, an empty record). When calling `add_id`, the argument is upcast by record subtyping to the parameter type, forgetting that the field `id` existed and was a string. This program is "statically typed" but unsafe to run, and is an example that record subtyping and field mutation together always require run-time type enforcement, so they should never be considered a "statically typed" combination: at best they can be gradual, and at worst they are unsafe.

6. The length of debugging sessions was validated when prototyping the experiment.

7. In $\text{GTFL}_{\lesssim}$, this type difference would be sufficient to make the program fail. But TypeScript, as an extension of JavaScript, relies on implicit conversions and could convert the string "2000" to the number 2000 whenever needed and continue evaluation. This behaviour can lead to unexpected results, harder to debug than what I intend to evaluate in this study. To ensure this program always fails in TypeScript, I go a step further and replace the zeros in the string with the capital O letter. The string "2OOO" in the first line cannot then be implicitly converted by TypeScript into a number.

8. The failure of interest in the original publication relies on lazy contract enforcement for mutable references in Racket (`box/c`). The lazy contract enforcement of `box/c` impacts the contract checking order, and that breaks one of the assumptions for the "blame shifting" debugging technique. I replaced the `box/c` contracts with higher order types. The original problem does not arise in AGT because this kind of delayed computation in the run-time type representation does not occur.

9. Caine (2016) identifies that 70% of the user studies published at CHI 2014 reported a sample size of less than 30, and that the most common sample size is 12. The author also establishes standards for sample sizes in HCI. Although prospective power analysis is a formal and statistically defensible method to determine the sample size, the author reports that this method is limited in the context of innovative technology as it requires preliminary data that often does not exist, and discusses alternative approaches to justify sample sizes. In general, it is also the case that remote studies tend to include more participants than in-person studies. However, justifying sample sizes remains an uncommon practice for both quantitative and qualitative studies in HCI, as reported by Salehzadeh Niksirat et al. (2023).

7 CONCLUSION

Abstracting Gradual Typing provides a systematic approach for the design of gradually typed languages. This dissertation discusses two limitations of the framework. First, the original definition of AGT does not enforce all expected modular type-based semantic invariants at run time. Second, the original design does not provide any detailed feedback to programmers when a run-time check fails. The first situation arises when language designers make a mistake in the design of their run-time type enforcement abstractions, and the second situations arises whenever language users make a mistake in their programs.

My thesis addresses these two issues. I constrain the AGT framework to ensure precise abstractions that enforce all type invariants at run time. I propose program slicing as a feedback tool for programmers when a program runs to a run-time type error. I discuss program slicing in a systematic way using Galois slicing and provide empirical evidence that program slices can be useful to developers facing a run-time type error.

In Chapter 3, I discuss precise abstractions for AGT. I begin by introducing the AGT framework by example, obtaining in the process a gradual version of the simply typed lambda calculus, AGTLC. This language provides a simple context to discuss the principles and requirements for the application of AGT, but it does not immediately exhibit the precision issues I intend to address. The issues already manifest in the $\text{GTFL}_{\lesssim}$ language introduced by Garcia, Clark, and Tanter (2016), which includes records and subtyping. I discuss programs that run successfully in this language even though they should produce a run-time error instead.

The key problem in this language manifests in the operation that propagates run-time type information, evidence composition. Because the original design of evidence composition is imprecise, some programs that should produce an error do not do so. I design a precise run-time abstraction for $\text{GTFL}_{\lesssim}$ in Section 3.4, and in Section 3.5, I discuss forward completeness, a general criteria from abstract interpretation that forces run-time abstractions to be precise. Precise run-time abstractions allow the language to enforce every type invariant present in the original program. Forward completeness serves as a guideline for future designs of run-time type abstractions and ensures that language designers do not mistakenly deploy imprecise abstractions moving forward.

In Chapter 4, I introduce Galois slicing (Perera, Acar, et al., 2012), a systematic approach for the design of slicing systems for programming languages. I presented Galois slicing in a simple context first, a calculator with a random number generator, and proceeded to address increasingly complex languages: adding functions, mutable state, and finally first-class continuations. With functions I address slicing substitutions, a key part of many programming language formalisms. This is the first work to explicitly discuss Galois slicing through substitutions instead of relying on environments. With mutable state, intermediate program state representations cannot be represented by program terms alone, and I discuss the slicing process interactions between program terms and the

store. With first-class continuations I show that Galois slicing scales to advanced control structures.

Galois slicing is a set of guidelines for the design of slicing systems. A forward slicing function must conservatively extend the original trace it is slicing, and monotonically map any possible slice of the inputs of the original trace to a slice of their outputs. When a forward slicing is well behaved for meets and joins (Definition 16), it uniquely determines a corresponding backward slicing function. Because Galois slicing systems compose, multi-step forward slicing can be systematically defined by repeatedly applying the single-step slicing system.

In Chapter 5, I apply Galois slicing to the AGT languages discussed in Chapter 3. In theses languages, slicing collects the type inconsistencies leading to a particular run-time failure and distinguishes parts of a program guaranteed not to be involved in a particular failure. These type inconsistencies cannot be removed by changing the run-time type information that is marked as irrelevant. When multiple inconsistencies arise, my design ensures that they are all preserved.

But we cannot provide slicing only for failures. To properly deal with failures, we also must provide slicing for successful computations. Because not all run-time failures are immediate, we must propagate successful information, which will reach a future evaluation step where the failure is to be detected. Programmers can use this slicing system to explore and understand the relationships between parts of a program trace, independently of the final evaluation result.

Designing a slicing system for an AGT language requires slicing the type-related operations and definitions in the runtime semantics. Similarly, the elaboration process converting source programs to a run-time representation must be sliced. Once one provides slicing systems for these relations, the ideas presented in Chapter 4 can be reused to provide a slicing system for multi-step reduction in an AGT language.

In Chapter 6, I provide empirical support for program slicing as a debugging tool in the context of run-time type errors. I prototyped an interface for a run-time error slicing tool and performed a Wizard of Oz user study, asking participants to highlight what they thought should be changed in the program, to describe a change recommendation based on their highlights, and to provide open feedback on the slicing tool.

Participants found the slicing tool useful. The overall score for a system usability scale survey was "good", and participants provided supporting positive open feedback. When negative feedback was present, it focused on the interface design and not on the information in the slices themselves. Slice information can alter the overall areas of focus for developers debugging a run-time type error, observable as a difference in program highlights. The study provides some evidence that developers relied on the slicing tool and that slices help programmers perceive what should be changed in a program.

The work in this area is far from over. Although my user study provides initial data, further research should explore whether our results generalize to broader developer populations. Our tool prototype can be improved with user-centred design approaches. Further research should explore the practical implications of slicing for developers in debugging run-time type errors, and compare this slicing approach with other gradual typing debugging approaches like blame. This comparison must be done both at the theoretical and at the practical level. Future research should explore whether new chal-

lenges arise when applying our slicing framework to other AGT language designs or to real-world languages. Participants in our study expressed interest in using this tool in real-world code. Providing real-world slicing requires research exploring low-overhead tracing approaches and the interactions between slicing and real-world language implementation subtleties. These all remain to be explored to provide a research agenda for the future of debugging for languages based on Abstracting Gradual Typing.

BIBLIOGRAPHY

Abadi, Martin, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin (1989). "Dynamic Typing in a Statically-Typed Language". In: *POPL*. DOI: 10.1145/75277.75296.

Acar, Umut A., Amal Ahmed, James Cheney, and Roly Perera (2013). "A Core Calculus for Provenance". In: *J. Cmput. Secur.* DOI: 10.3233/JCS-130487.

Ahmed, Amal, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler (Jan. 2011). "Blame for All". In: POPL '11, pp. 201–214. DOI: 10.1145/1926385.1926409.

Ahmed, Amal, Dustin Jamner, Jeremy G. Siek, and Philip Wadler (Aug. 2017). "Theorems for Free for Free: Parametricity, with and without Types". In: *Proc. ACM Program. Lang.* 1.ICFP. DOI: 10.1145/3110283.

Allende, Esteban, Johan Fabry, Ronald Garcia, and Éric Tanter (2014). "Confined Gradual Typing". In: OOPSLA '14, pp. 251–270. DOI: 10.1145/2660193.2660222.

Alwis, B. de and G.C. Murphy (2006). "Using Visual Momentum to Explain Disorientation in the Eclipse IDE". In: VL/HCC '06, pp. 51–54. DOI: 10.1109/VLHCC.2006.49.

Anderson, Christopher and Sophia Drossopoulou (2003). "BabyJ: From Object Based to Class Based Programming via Types". In: *ENTCS*. DOI: 10.1016/S1571-0661(04)80802-8.

Augustine (426a). *The City of God (De civitate Dei)*. Trans. by Marcus (1950) Dods. Random House. ISBN: 9780307764768.

Augustine (426b). *The City of God against the Pagans*. Trans. by R.W. (1998) Dyson. Cambridge University Press. ISBN: 9780521464758.

Bader, Johannes, Jonathan Aldrich, and Éric Tanter (2018). "Gradual Program Verification". In: VMCAI '17, pp. 25–46. DOI: 10.1007/978-3-319-73721-8_2.

Bañados Schwerter, Felipe (2016). "Side-effects take the blame". In: *SLE*. DOI: 10.1145/2997364.2997381.

Bañados Schwerter, Felipe, Alison M. Clark, Khurram A. Jafery, and Ronald Garcia (Jan. 2021). "Abstracting Gradual Typing Moving Forward: Precise and Space-Efficient". In: *Proc. ACM Program. Lang.* 5.POPL. DOI: 10.1145/3434342.

Bañados Schwerter, Felipe, Ronald Garcia, and Éric Tanter (2014). "A Theory of Gradual Effect Systems". In: *ICFP*. DOI: 10.1145/2628136.2628149.

Bañados Schwerter, Felipe, Ronald Garcia, and Éric Tanter (Sept. 2016). "Gradual type-and-effect systems". In: *JFP* 26. DOI: 10.1017/S0956796816000162.

Bangor, Aaron, Philip T. Kortum, and James T. Miller (2008). "An Empirical Evaluation of the System Usability Scale". In: *International Journal of Human–Computer Interaction* 24.6, pp. 574–594. DOI: 10.1080/10447310802205776.

Barik, Titus, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin (2017). "Do Developers Read Compiler Error Messages?" In: ICSE '17, pp. 575–585. DOI: 10.1109/ICSE.2017.59.

Bauman, Spenser, Carl Friedrich Bolz-Tereick, Jeremy G. Siek, and Sam Tobin-Hochstadt (Oct. 2017). "Sound Gradual Typing: Only Mostly Dead". In: *Proc. ACM Program. Lang.* 1.OOPSLA. DOI: 10.1145/3133878.

Becker, Brett A. et al. (2019). "Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research". In: ITiCSE-WGR '19, pp. 177–210. DOI: 10.1145/3344429.3372508.

Beugnard, A., J.-M. Jezequel, N. Plouzeau, and D. Watkins (1999). "Making components contract aware". In: *Computer* 32.7, pp. 38–45. DOI: 10.1109/2.774917.

Blume, Matthias and David McAllester (2006). "Sound and complete models of contracts". In: *JFP* 16 (4-5), pp. 375–414. DOI: 10.1017/S0956796806005971.

Bracha, Gilad (2004). "Pluggable type systems". In: *OOPSLA'04 Workshop on Revival of Dynamic Languages*. URL: http://www.bracha.org/pluggableTypesPosition.pdf.

Brooke, John (1996). "SUS: a 'quick and dirty' usability scale". In: *Usability Evaluation in Industry*. 1st ed. CRC Press, p. 6.

Brooke, John (Feb. 2013). "SUS: A Retrospective". In: *Journal of User Experience* 8.2, pp. 29–40. URL: https://uxpajournal.org/sus-a-retrospective/.

Brooks, Frederick P. (1975). *The Mythical Man-Month*. Addison-Wesley.

Buneman, Peter, Sanjeev Khanna, and Wang-chiew Tan (2001). "Why and Where: A Characterization of Data Provenance". In: *In ICDT*. Springer, pp. 316–330.

Caine, Kelly (2016). "Local Standards for Sample Size at CHI". In: CHI '16, pp. 981–992. DOI: 10.1145/2858036.2858498.

Campora, John Peter, Sheng Chen, and Eric Walkingshaw (July 2018). "Casts and Costs: Harmonizing Safety and Performance in Gradual Typing". In: *Proc. ACM Program. Lang.* 2.ICFP. DOI: 10.1145/3236793.

Carata, Lucian, Sherif Akoush, Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, Margo Seltzer, and Andy Hopper (May 2014). "A Primer on Provenance". In: *CACM* 57.5, pp. 52–60. DOI: 10.1145/2596628.

Cardelli, Luca (1997). "Type Systems". In: *Handbook of Computer Science and Engineering*. URL: http://lucacardelli.name/Papers/TypeSystems.pdf.

Cartwright, Robert and Mike Fagan (1991). "Soft Typing". In: PLDI '91, pp. 278–292. DOI: 10.1145/113445.113469.

Castagna, Giuseppe and Victor Lanvin (Aug. 2017). "Gradual Typing with Union and Intersection Types". In: *Proc. ACM Program. Lang.* 1.ICFP. DOI: 10.1145/3110285. URL: https://doi.org/10.1145/3110285.

Chen, Sheng and John Peter Campora III (2019). "Blame Tracking and Type Error Debugging". In: vol. 136. SNAPL '19, 2:1–2:14. DOI: 10.4230/LIPIcs.SNAPL.2019.2.

Chen, Zhifei, Yanhui Li, Bihuan Chen, Wanwangying Ma, Lin Chen, and Baowen Xu (2020). "An Empirical Study on Dynamic Typing Related Practices in Python Systems". In: ICPC '20, pp. 83–93. DOI: 10.1145/3387904.3389253.

Cheney, James, Laura Chiticariu, and Wang-Chiew Tan (2009). "Provenance in Databases: Why, How, and Where". In: *Foundations and Trends in Databases* 1.4, pp. 379–474. DOI: 10.1561/1900000006.

Choppella, Venkatesh and Christopher T. Haynes (2003). "Source-Tracking Unification". In: CADE '03, pp. 458–472. DOI: 10.1007/978-3-540-45085-6_39.

Church, Alonzo (Apr. 1932). "A Set of Postulates for the Foundation of Logic". In: *Annals of Mathematics*. Second Series 33.2, pp. 346–366. DOI: 10.2307/1968337.

Church, Alonzo (June 1940). "A formulation of the simple theory of types". In: *Journal of Symbolic Logic* 5.2, pp. 56–68. DOI: 10.2307/2266170.

Church, Alonzo (1941). *The Calculi of Lambda-Conversion*. Annals of Mathematics Studies 6. Princeton: Princeton University Press.

Coblenz, Michael, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine (Nov. 2020). "Can Advanced Type Systems Be Usable? An Empirical Study of Ownership, Assets, and Typestate in Obsidian". In: *Proc. ACM Program. Lang.* 4.OOPSLA. DOI: 10.1145/3428200.

Coblenz, Michael, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers (July 2021). "PLIERS: A Process That Integrates User-Centered Methods into Programming Language Design". In: *ACM Trans. Comput.-Hum. Interact.* 28.4. DOI: 10.1145/3452379.

Cousot, Patrick and Radhia Cousot (1977). "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: POPL '77, pp. 238–252. DOI: 10.1145/512950.512973.

Cui, Yingwei, Jennifer Widom, and Janet L. Wiener (June 2000). "Tracing the Lineage of View Data in a Warehousing Environment". In: *TODS* 25.2, pp. 179–227. DOI: 10.1145/357775.357777.

Danas, Natasha, Tim Nelson, Lane Harrison, Shriram Krishnamurthi, and Daniel J. Dougherty (2017). "User Studies of Principled Model Finder Output". In: SEFM '17, pp. 168–184. DOI: 10.1007/978-3-319-66197-1_11.

Dell, Nicola, Vidya Vaidyanathan, Indrani Medhi, Edward Cutrell, and William Thies (2012). ""Yours is Better!": Participant Response Bias in HCI". In: CHI '12, pp. 1321–1330. DOI: 10.1145/2207676.2208589.

Deursen, A. van, T. B. Dinesh, and E. A. van der Meulen (1994). "The ASF+SDF Meta-environment". In: AMAST '93, pp. 411–412. DOI: 10.1007/978-1-4471-3227-1_47.

Dimoulas, Christos (2012). "Foundations for Behavioral Higher-Order Contracts". PhD thesis. Northeastern University. DOI: 10.17760/d20002848.

Dimoulas, Christos and Matthias Felleisen (Nov. 2011). "On Contract Satisfaction in a Higher-order World". In: *TOPLAS* 33.5. DOI: 10.1145/2039346.2039348.

Dimoulas, Christos, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen (2011). "Correct Blame for Contracts: No More Scapegoating". In: POPL '11. DOI: 10.1145/1926385.1926410.

Dimoulas, Christos, Max S. New, Robert Bruce Findler, and Matthias Felleisen (2016). "Oh Lord, Please Don'T Let Contracts Be Misunderstood (Functional Pearl)". In: ICFP '16. DOI: 10.1145/2951913.2951930.

Dimoulas, Christos, Sam Tobin-Hochstadt, and Matthias Felleisen (2012). "Complete Monitors for Behavioral Contracts". In: *ESOP*. DOI: 10.1007/978-3-642-28869-2_11.

Erdweg, Sebastian et al. (2015). "Evaluating and comparing language workbenches: Existing results and benchmarks for the future". In: *Computer Languages, Systems & Structures* 44, pp. 24–47. DOI: 10.1016/j.cl.2015.08.007.

Eremondi, Joseph (2023). "On the design of a gradual dependently typed language for programming". PhD thesis. University of British Columbia. DOI: http://dx.doi.org/10.14288/1.0428823.

Eremondi, Joseph, Wouter Swierstra, and Jurriaan Hage (Jan. 2019). "A framework for improving error messages in dependently-typed languages". In: *Open Computer Science* 9.1, pp. 1–32. DOI: 10.1007/978-3-642-28869-2_11.

Eremondi, Joseph, Éric Tanter, and Ronald Garcia (July 2019). "Approximate Normalization for Gradual Dependent Types". In: *Proc. ACM Program. Lang.* 3.ICFP. DOI: 10.1145/3341692.

Feltey, Daniel, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour (Oct. 2018). "Collapsible Contracts: Fixing a Pathology of Gradual Typing". In: *Proc. ACM Program. Lang.* 2.OOPSLA. DOI: 10.1145/3276503.

Field, John, G. Ramalingam, and Frank Tip (1995). "Parametric Program Slicing". In: POPL '95, pp. 379–392. DOI: 10.1145/199448.199534.

Field, John and Frank Tip (1998). "Dynamic dependence in term rewriting systems and its application to program slicing". In: *Information and Software Technology* 40.11, pp. 609–636. DOI: 10.1016/S0950-5849(98)00087-1.

Findler, Robert Bruce and Matthias Blume (2006). "Contracts as Pairs of Projections". In: *Functional and Logic Programming*, pp. 226–241. DOI: 10.1007/11737414_16.

Findler, Robert Bruce and Matthias Felleisen (2001). "Contract Soundness for Object-Oriented Languages". In: OOPSLA '01, pp. 1–15. DOI: 10.1145/504282.504283.

Findler, Robert Bruce and Matthias Felleisen (2002). "Contracts for Higher-Order Functions". In: ICFP '02. DOI: 10.1145/581478.581484.

Findler, Robert Bruce, Mario Latendresse, and Matthias Felleisen (2001). "Behavioral Contracts and Behavioral Subtyping". In: FSE '01, pp. 229–236. DOI: 10.1145/503209.503240.

Floyd, Robert W. (1967). "Assigning Meanings to Programs". In: *Proceedings of Symposium on Applied Mathematics* 19, pp. 19–32.

Gao, Zheng, Christian Bird, and Earl T Barr (2017). "To type or not to type: quantifying detectable bugs in JavaScript". In: ICSE '17, pp. 758–769. DOI: 10.1109/ICSE.2017.75.

Garcia, Ronald (2013). "Calculating Threesomes, with Blame". In: *ICFP*. DOI: 10.1145/2500365.2500603.

Garcia, Ronald and Matteo Cimini (2015). "Principal Type Schemes for Gradual Programs". In: POPL '15, pp. 303–315. DOI: 10.1145/2676726.2676992.

Garcia, Ronald, Alison M. Clark, and Éric Tanter (2016). "Abstracting Gradual Typing". In: *POPL*. DOI: 10.1145/2837614.2837670.

Garcia, Ronald and Éric Tanter (2020). "Gradual Typing as if Types Mattered". In: *WGT*. URL: https://wgt20.irif.fr/wgt20-final28-acmpaginated.pdf.

Gentzen, Gerhard (1932). "Über die Existenz unabhängiger Axiomensysteme zu unendlichen Satzsystemen". In: *Mathematische Annalen* 107.1, pp. 329–350. DOI: 10.1007/BF01448897.

Gentzen, Gerhard (1969). "On the Existence of Independent Axiom Systems for Infinite Sentence Systems". In: *The Collected Papers of Gerhard Gentzen*. Vol. 55, pp. 29–52. DOI: 10.1016/S0049-237X(08)70820-6.

Giacobazzi, Roberto and Elisa Quintarelli (2001). "Incompleteness, Counterexamples, and Refinements in Abstract Model-Checking". In: SAS '01, pp. 356–373. URL: http://dl.acm.org/citation.cfm?id=647170.718288.

Gierczak, Olek, Lucy Menon, Christos Dimoulas, and Amal Ahmed (2024). "Gradually Typed Languages Should Be Vigilant!" In: OOPSLA. DOI: 10.1145/3649842.

Green, Thomas R. G. (1990). "The Cognitive Dimension of Viscosity: A Sticky Problem for HCI". In: INTERACT '90, pp. 79–86. ISBN: 0444888179.

Green, Todd J., Grigoris Karvounarakis, and Val Tannen (2007). "Provenance Semirings". In: *PODS*. DOI: 10.1145/1265530.1265535.

Greenberg, Michael (2019). "The Dynamic Practice and Static Theory of Gradual Typing". In: SNAPL '19, 6:1–6:20. DOI: 10.4230/LIPIcs.SNAPL.2019.6.

Greenman, Ben (2020). "Deep and shallow types". PhD thesis. Northeastern University. DOI: 10.17760/D20398329.

Greenman, Ben, Christos Dimoulas, and Matthias Felleisen (Mar. 2023). "Typed–Untyped Interactions: A Comparative Analysis". In: *TOPLAS* 45.1. DOI: 10.1145/3579833.

Greenman, Ben and Matthias Felleisen (July 2018). "A Spectrum of Type Soundness and Performance". In: *Proc. ACM Program. Lang.* 2.ICFP. DOI: 10.1145/3236766.

Greenman, Ben, Matthias Felleisen, and Christos Dimoulas (2019). "Complete Monitors for Gradual Types". In: OOPSLA. DOI: 10.1145/3360548.

Haack, Christian and J.B. Wells (2004). "Type error slicing in implicitly typed higher-order languages". In: *SCP* 50.1. ESOP '03, pp. 189–224. DOI: 10.1016/j.scico.2004.01.004.

Hage, Jurriaan and Bastiaan Heeren (2007). "Heuristics for Type Error Discovery and Recovery". In: *Implementation and Application of Functional Languages*, pp. 199–216. DOI: 10.1007/978-3-540-74130-5_12.

Hanenberg, Stefan, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik (Oct. 2014). "An Empirical Study on the Impact of Static Typing on Software Maintainability". In: *Empirical Software Engineering*. DOI: 10.1007/s10664-013-9289-1.

Harper, Robert (2013). *Practical Foundations for Programming Languages*. Cambridge University Press.

Hertz, Paul (Sept. 1922). "Über Axiomensysteme für beliebige Satzsysteme. I. Teil. Sätze ersten Grades. (Über die Axiomensysteme von der kleinsten Satzzahl und den Begriff des idealen Elementes.)" In: *Mathematische Annalen* 87.3, pp. 246–269. DOI: 10.1007/BF01459067.

Hertz, Paul (2012). "On Axiomatic Systems for Arbitrary Systems of Sentences, Part I, Sentences of the First Degree. (On Axiomatic Systems of the Smallest Number of Sentences and the Concept of the Ideal Element)". In: *Universal Logic: An Anthology, From Paul Hertz to Dov Gabbay*. Ed. by Jean-Yves Béziau. Studies in Universal Logic. Birkhäuser Basel.

Hoare, C. A. R. (Oct. 1969). "An Axiomatic Basis for Computer Programming". In: *CACM* 12.10, pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259.

Hoeflich, Joshua, Robert Bruce Findler, and Manuel Serrano (Oct. 2022). "Highly Illogical, Kirk: Spotting Type Mismatches in the Large despite Broken Contracts, Unsound Types, and Too Many Linters". In: *Proc. ACM Program. Lang.* 6.OOPSLA2. DOI: 10.1145/3563305.

Hu, Jingmei, Jiwon Joung, Maia Jacobs, Krzysztof Z. Gajos, and Margo I. Seltzer (2020). "Improving Data Scientist Efficiency with Provenance". In: ICSE '20, pp. 1086–1097. DOI: 10.1145/3377811.3380366.

Ikeda, R., J. Cho, C. Fang, S. Salihoglu, S. Torikai, and J. Widom (2012). "Provenance-Based Debugging and Drill-Down in Data-Oriented Workflows". In: *2012 IEEE 28th International Conference on Data Engineering*. DOI: 10.1109/ICDE.2012.118.

Ingerman, Peter Zilahy (Mar. 1967). ""Pānini-Backus Form" Suggested". In: *CACM* 10.3, p. 137. DOI: 10.1145/363162.363165.

Jacobs, Koen, Amin Timany, and Dominique Devriese (Jan. 2021). "Fully Abstract from Static to Gradual". In: *Proc. ACM Program. Lang.* 5.POPL. DOI: 10.1145/3434288.

Jafery, Khurram A. and Jana Dunfield (2017). "Sums of Uncertainty: Refinements Go Gradual". In: POPL '17. DOI: 10.1145/3009837.3009865.

Jakubovic, Joel, Jonathan Edwards, and Tomas Petricek (2023). "Technical Dimensions of Programming Systems". In: *The Art, Science, and Engineering of Programming* 7 (3). DOI: 10.22152/programming-journal.org/2023/7/13.

Jia, Limin, Hannah Gommerstadt, and Frank Pfenning (2016). "Monitors and Blame Assignment for Higher-order Session Types". In: *POPL*. DOI: 10.1145/2837614.2837662.

Johnson, Andrew, Lucas Waye, Scott Moore, and Stephen Chong (2015). "Exploring and Enforcing Security Guarantees via Program Dependence Graphs". In: PLDI '15. DOI: 10.1145/2737924.2737957.

Kelley, J. F. (1983). "An Empirical Methodology for Writing User-Friendly Natural Language Computer Applications". In: CHI '83, pp. 193–196. DOI: 10.1145/800045.801609.

Kierkegaard, Søren (1843). *Journal JJ*. Ed. by Bruce H. Kirmmse, K. Brian Söderquist, Niels Jørgen Cappelørn, Alastair Hannay, David Kangas, Vanessa Rumble, and George Pattison. Vol. 2. Kierkegaard's Journals and Notebooks. Princeton: Princeton University Press. ISBN: 9781400874330. DOI: doi:10.1515/9781400874330. URL: https://doi.org/10.1515/9781400874330.

Kleinschmager, Sebastian, Romain Robbes, Andreas Stefik, Stefan Hanenberg, and Éric Tanter (2012). "Do static type systems improve the maintainability of software systems? An empirical study". In: ICPC '12, pp. 153–162. DOI: 10.1109/ICPC.2012.6240483.

Knuth, Donald E. (Dec. 1974). "Computer Programming as an Art". In: *CACM* 17.12, pp. 667–673. DOI: 10.1145/361604.361612.

Knuth, Donald E. and Luis Trabb Pardo (1980). "The Early Development of Programming Languages". In: *A History of Computing in the Twentieth Century*, pp. 197–273. DOI: 10.1016/B978-0-12-491650-0.50019-8.

Ko, Amy J. and Brad A. Myers (2004). "Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior". In: CHI '04, pp. 151–158. DOI: 10.1145/985692.985712.

Ko, Amy J. and Brad A. Myers (2010). "Extracting and Answering Why and Why Not Questions About Java Program Output". In: *TOSEM*. DOI: 10.1145/1824760.1824761.

Ko, Amy J., Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung (2006). "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks". In: *TSE* 32.12, pp. 971–987. DOI: 10.1109/TSE.2006.116.

Korel, Bogdan and Janusz Laski (1988). "Dynamic program slicing". In: *Information Processing Letters* 29.3, pp. 155–163. DOI: 10.1016/0020-0190(88)90054-3.

Labrada, Elizabeth, Matías Toro, and Éric Tanter (Oct. 2022). "Gradual System F". In: *JACM* 69.5. DOI: 10.1145/3555986.

Labrada, Elizabeth, Matías Toro, Éric Tanter, and Dominique Devriese (Apr. 2022). "Plausible Sealing for Gradual Parametricity". In: *Proc. ACM Program. Lang.* 6.OOPSLA1. DOI: 10.1145/3527314.

Lazarek, Lukas, Ben Greenman, Matthias Felleisen, and Christos Dimoulas (Aug. 2021). "How to Evaluate Blame for Gradual Types". In: *Proc. ACM Program. Lang.* 5.ICFP. DOI: 10.1145/3473573.

Lazarek, Lukas, Alexis King, Samanvitha Sundar, Robert Bruce Findler, and Christos Dimoulas (Dec. 2019). "Does Blame Shifting Work?" In: *Proc. ACM Program. Lang.* 4.POPL. DOI: 10.1145/3371133.

Lehmann, Nico and Éric Tanter (2017). "Gradual Refinement Types". In: POPL '17, pp. 775–788. DOI: 10.1145/3009837.3009856.

Lennon-Bertrand, Meven, Kenji Maillard, Nicolas Tabareau, and Éric Tanter (Apr. 2022). "Gradualizing the Calculus of Inductive Constructions". In: *ACM Trans. Program. Lang. Syst.* 44.2. DOI: 10.1145/3495528.

Leroy, Xavier and Michel Mauny (1991). "Dynamics in ML". In: FPCA '91, pp. 406–426. DOI: 10.1007/3540543961_20.

Liskov, Barbara and Stephen Zilles (1974). "Programming with Abstract Data Types". In: *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pp. 50–59. DOI: 10.1145/800233.807045.

Lu, Kuang-Chen, Ben Greenman, Carl Meyer, Dino Viehland, Aniket Panse, and Shriram Krishnamurthi (2023). "Gradual Soundness: Lessons from Static Python". In: *The Art, Science, and Engineering of Programming* 7.1. Article 2. DOI: 10.22152/programming-journal.org/2023/7/2.

Lubin, Justin and Sarah E. Chasins (Oct. 2021). "How Statically-Typed Functional Programmers Write Code". In: *Proc. ACM Program. Lang.* 5.OOPSLA. DOI: 10.1145/3485532.

Malewski, Stefan, Michael Greenberg, and Éric Tanter (Oct. 2021). "Gradually Structured Data". In: *Proc. ACM Program. Lang.* 5.OOPSLA. DOI: 10.1145/3485503.

Marceau, Guillaume, Kathi Fisler, and Shriram Krishnamurthi (2011). "Mind Your Language: On Novices' Interactions with Error Messages". In: *Onward!* DOI: 10.1145/2048237.2048241.

Marino, Daniel and Todd Millstein (2009). "A generic type-and-effect system". In: *TLDI*, pp. 39–50. URL: https://web.cs.ucla.edu/~todd/research/tldi09.pdf.

Marques, Arthur (2022). "Supporting a developer's discovery of task-relevant information". PhD thesis. University of British Columbia. DOI: 10.14288/1.0421314.

Mayer, Clemens, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik (Oct. 2012). "An Empirical Study of the Influence of Static Type Systems on the Usability of Undocumented Software". In: *SIGPLAN Not.* 47.10, pp. 683–702. ISSN: 0362-1340. DOI: 10.1145/2398857.2384666. URL: https://doi.org/10.1145/2398857.2384666.

McCarthy, John (Dec. 1957). *A proposal for a compiler*. Memo CC-56. MIT, p. 19. URL: https://www.softwarepreservation.com/projects/LISP/MIT/McCarthy-CC-56.pdf.

McCarthy, John (Apr. 1960). "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". In: *CACM* 3.4, pp. 184–195. DOI: 10.1145/367177.367199.

McCarthy, John (1978). "History of LISP". In: *History of Programming Languages*, pp. 173–185. DOI: 10.1145/800025.1198360.

Menzies, Tim, William Nichols, Forrest Shull, and Lucas Layman (Aug. 2017). "Are delayed issues harder to resolve? Revisiting cost-to-fix of defects throughout the

lifecycle". In: *Empirical Software Engineering* 22.4, pp. 1903–1935. DOI: 10.1007/s10664-016-9469-x.

Meyer, B. (Oct. 1992). "Applying 'design by contract'". In: *IEEE Computer* 25.10, pp. 40–51. DOI: 10.1109/2.161279.

Milner, Robin (1978). "A theory of type polymorphism in programming". In: *JCSS*, pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4.

Moy, Cameron, Phúc C. Nguyundefinedn, Sam Tobin-Hochstadt, and David Van Horn (Jan. 2021). "Corpse Reviver: Sound and Efficient Gradual Typing via Contract Verification". In: *Proc. ACM Program. Lang.* 5.POPL. DOI: 10.1145/3434334.

Muehlboeck, Fabian and Ross Tate (Oct. 2017). "Sound Gradual Typing is Nominally Alive and Well". In: *Proc. ACM Program. Lang.* 1.OOPSLA. DOI: 10.1145/3133880.

Munkres, James R. (2000). *Topology: Second Edition*. Upper Saddle River, NJ: Prentice Hall.

Myers, Brad A., Amy J. Ko, T. D. LaToza, and Y. Yoon (July 2016). "Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools". In: *IEEE Computer* 49.7, pp. 44–52. DOI: 10.1145/2851581.2886434.

Myers, Brad A., Andreas Stefik, Stefan Hanenberg, Antti-Juhani Kaijanaho, Margaret Burnett, Franklyn Turbak, and Philip Wadler (2016). "Usability of Programming Languages: Special Interest Group (SIG) Meeting at CHI 2016". In: *CHI EA*. ACM. DOI: 10.1145/2851581.2886434.

Naur, Peter and Brian Randell, eds. (Oct. 1968). *SOFTWARE ENGINEERING: Report on a conference sponsored by the NATO science committee*. Brussels, Belgium: Scientific Affairs Division, NATO.

Nelson, Tim, Natasha Danas, Daniel J. Dougherty, and Shriram Krishnamurthi (2017). "The Power of "Why" and "Why Not": Enriching Scenario Exploration with Provenance". In: *ESEC/FSE*. DOI: 10.1145/3106237.3106272.

New, Max S. and Amal Ahmed (July 2018). "Graduality from Embedding-Projection Pairs". In: *Proc. ACM Program. Lang.* 2.ICFP. DOI: 10.1145/3236768.

New, Max S., Dustin Jamner, and Amal Ahmed (Jan. 2020). "Graduality and Parametricity: Together Again for the First Time". In: *Proc. ACM Program. Lang.* 4.POPL. DOI: 10.1145/3371114.

New, Max S., Daniel R. Licata, and Amal Ahmed (Jan. 2019). "Gradual Type Theory". In: *Proc. ACM Program. Lang.* 3.POPL. DOI: 10.1145/3290328.

Ore, John-Paul, Carrick Detweiler, and Sebastian Elbaum (Feb. 2021). "An Empirical Study on Type Annotations: Accuracy, Speed, and Suggestion Effectiveness". In: *TOSEM* 30.2. DOI: 10.1145/3439775.

Ore, Oystein (May 1944). "Galois connexions". In: *Transactions of the American Mathematical Society* 55, pp. 493–513. DOI: 10.2307/1990305.

Perera, Roly, Umut A. Acar, James Cheney, and Paul Blain Levy (2012). "Functional Programs That Explain Their Work". In: *ICFP*. DOI: 10.1145/2364527.2364579.

Perera, Roly, Deepak Garg, and James Cheney (2016). "Causally Consistent Dynamic Slicing". In: vol. 59. CONCUR '16, 18:1–18:15. DOI: 10.4230/LIPIcs.CONCUR.2016.18.

Perlis, Alan J. (1978). "The American Side of the Development of ALGOL". In: *History of Programming Languages*, pp. 75–91. DOI: 10.1145/800025.1198352.

Perlis, Alan J. (Sept. 1982). "Special Feature: Epigrams on Programming". In: *SIGPLAN Not.* 17.9, pp. 7–13. DOI: 10.1145/947955.1083808.

Pierce, Benjamin C, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey (2019). "Programming Language Foundations". In: *Software Foundations*. Ed. by Benjamin C Pierce. https://softwarefoundations.cis.upenn.edu/plf-current/.

Pierce, Benjamin C. (2002). *Types and programming languages*. Cambridge, MA, USA: The MIT Press.

Pimentel, João Felipe, Leonardo Murta, Vanessa Braganholo, and Juliana Freire (Aug. 2017). "noWorkflow: A Tool for Collecting, Analyzing, and Managing Provenance from Python Scripts". In: *Proc. VLDB Endow.* 10.12. DOI: 10.14778/3137765.3137789.

Plotkin, Gordon D. (1981). *A structural approach to operational semantics*. DAIMI FN-19. Aarhus University.

Politz, Joe Gibbs, Hannah Quay-de la Vallee, and Shriram Krishnamurthi (2012). "Progressive Types". In: Onward! 2012, pp. 55–66. DOI: 10.1145/2384592.2384599.

Prather, James, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen (2017). "On Novices' Interaction with Compiler Error Messages: A Human Factors Approach". In: ICER '17. DOI: 10.1145/3105726.3106169.

Ramsey, F. P. (1926). "The Foundations of Mathematics". In: *Proceedings of the London Mathematical Society* s2-25.1, pp. 338–384. DOI: 10.1112/plms/s2-25.1.338.

Rastogi, Aseem, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris (2015). "Safe & Efficient Gradual Typing for TypeScript". In: *POPL*. DOI: 10.1145/2676726.2676971.

Remington Rand Univac (1958). *UNIVAC math-matic programming system*. Tech. rep. Remington Rand Univac. URL: http://bitsavers.org/pdf/univac/univac2/U-1568_MATHMATIC_PgmgSys_1958.pdf.

Rémy, D. (1989). "Type Checking Records and Variants in a Natural Extension of ML". In: *POPL*, pp. 77–88. DOI: 10.1145/75277.75284.

Ricciotti, Wilmer, Jan Stolarek, Roly Perera, and James Cheney (Aug. 2017). "Imperative Functional Programs That Explain Their Work". In: ICFP. DOI: 10.1145/3110258.

Richards, Gregor, Ellen Arteca, and Alexi Turcotte (Oct. 2017). "The VM Already Knew That: Leveraging Compile-time Knowledge to Optimize Gradual Typing". In: *Proc. ACM Program. Lang.* 1.OOPSLA. DOI: 10.1145/3133879.

Richards, Gregor, Francesco Zappa Nardelli, and Jan Vitek (2015). "Concrete Types for TypeScript". In: vol. 37. ECOOP '15, pp. 76–100. DOI: 10.4230/LIPIcs.ECOOP.2015.76.

Rodríguez-Pérez, Gema, Andy Zaidman, Alexander Serebrenik, Gregorio Robles, and Jesús M. González-Barahona (2018). "What If a Bug Has a Different Origin? Making Sense of Bugs without an Explicit Bug Introducing Change". In: ESEM '18. DOI: 10.1145/3239235.3267436.

Russell, Bertrand (1903). *The Principles of Mathematics*. Cambridge University Press.

Salehzadeh Niksirat, Kavous, Lahari Goswami, Pooja S. B. Rao, James Tyler, Alessandro Silacci, Sadiq Aliyu, Annika Aebli, Chat Wacharamanotham, and Mauro Cherubini (2023). "Changes in Research Ethics, Openness, and Transparency in Empirical Studies between CHI 2017 and CHI 2022". In: CHI '23. DOI: 10.1145/3544548.3580848.

Seidel, Eric L., Ranjit Jhala, and Westley Weimer (2016). "Dynamic Witnesses for Static Type Errors (or, Ill-typed Programs Usually Go Wrong)". In: *ICFP*. DOI: 10.1145/2951913.2951915.

Sergey, Ilya and Dave Clarke (2012). "Gradual Ownership Types". In: *ESOP*, pp. 579–599. DOI: 10.1007/978-3-642-28869-2_29.

Siek, Jeremy G., Ronald Garcia, and Walid Taha (2009). "Exploring the Design Space of Higher-Order Casts". In: *ESOP*. DOI: 10.1007/978-3-642-00590-9_2.

Siek, Jeremy G. and Walid Taha (2006). "Gradual Typing for Functional Languages". In: *SFP*, pp. 81–92. URL: http://scheme2006.cs.uchicago.edu/13-siek.pdf.

Siek, Jeremy G. and Walid Taha (2007). "Gradual typing for objects". In: *ECOOP '07*, pp. 2–27. DOI: 10.1007/978-3-540-73589-2_2.

Siek, Jeremy G., Peter Thiemann, and Philip Wadler (2015). "Blame and Coercion: Together Again for the First Time". In: *PLDI*, pp. 425–435. DOI: 10.1145/2737924.2737968.

Siek, Jeremy G., Michael M. Vitousek, Matteo Cimini, and John Tang Boyland (2015). "Refined Criteria for Gradual Typing". In: SNAPL '15. DOI: 10.4230/LIPIcs.SNAPL.2015.274.

Siek, Jeremy G. and Philip Wadler (2010). "Threesomes, with and without blame". In: *POPL*. DOI: 10.1145/1706299.1706342.

Steele, Guy L. (2017). "It's Time for a New Old Language". In: PPoPP '17. Austin, Texas, USA, p. 1. DOI: 10.1145/3018743.3018773.

Stuckey, Peter J., Martin Sulzmann, and Jeremy Wazny (2003). "Interactive Type Debugging in Haskell". In: Haskell '03, pp. 72–83. DOI: 10.1145/871895.871903.

Stuckey, Peter J., Martin Sulzmann, and Jeremy Wazny (2004). "Improving Type Error Diagnosis". In: Haskell '04, pp. 80–91. DOI: 10.1145/1017472.1017486.

Takikawa, Asumu, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen (2016). "Is Sound Gradual Typing Dead?" In: *POPL*. DOI: 10.1145/2837614.2837630.

Takikawa, Asumu, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen (2012). "Gradual Typing for First-Class Classes". In: *OOPSLA*, pp. 793–810. DOI: 10.1145/2384616.2384674.

Thatte, Satish (1989). "Quasi-Static Typing". In: POPL '90, pp. 367–381. DOI: 10.1145/96709.96747.

Tip, Frank (1995). "Generic techniques for source-level debugging and dynamic program slicing". In: TAPSOFT '95, pp. 516–530. DOI: 10.1007/3-540-59293-8_217.

Tip, Frank and T. B. Dinesh (Jan. 2001). "A Slicing-Based Approach for Locating Type Errors". In: *TOSEM* 10.1, pp. 5–55. DOI: 10.1145/366378.366379.

Tobin-Hochstadt, Sam and Matthias Felleisen (2006). "Interlanguage Migration: From Scripts to Programs". In: *OOPSLA*. DOI: 10.1145/1176617.1176755.

Tobin-Hochstadt, Sam and Matthias Felleisen (2008). "The Design and Implementation of Typed Scheme". In: *POPL*.

Tobin-Hochstadt, Sam, Matthias Felleisen, Robert Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa (2017). "Migratory Typing: Ten Years Later". In: vol. 71. SNAPL '17, 17:1–17:17. DOI: 10.4230/LIPIcs.SNAPL.2017.17.

Toro, Matías, Ronald Garcia, and Éric Tanter (Dec. 2018). "Type-Driven Gradual Security with References". In: *TOPLAS* 40.4. DOI: 10.1145/3229061.

Toro, Matías, Elizabeth Labrada, and Éric Tanter (2019). "Gradual Parametricity, Revisited". In: POPL. DOI: 10.1145/3290330.

Toro, Matías and Éric Tanter (2017). "A Gradual Interpretation of Union Types". In: pp. 382–404. DOI: 10.1007/978-3-319-66706-5_19.

Toro, Matías and Éric Tanter (2020). "Abstracting gradual references". In: *SCP* 197, p. 102496. DOI: 10.1016/j.scico.2020.102496.

Tunnell Wilson, Preston, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi (2018). "The Behavior of Gradual Types: A User Study". In: *DLS*. DLS '18, pp. 1–12. DOI: 10.1145/3276945.3276947.

van Deursen, A., P. Klint, and Frank Tip (1993). "Origin tracking". In: *Journal of Symbolic Computation* 15.5, pp. 523–545. DOI: 10.1016/S0747-7171(06)80004-0.

Vazou, Niki, Éric Tanter, and David Van Horn (Oct. 2018). "Gradual Liquid Type Inference". In: *Proc. ACM Program. Lang.* 2.OOPSLA. DOI: 10.1145/3276502.

Vitousek, Michael M., Cameron Swords, and Jeremy G. Siek (2017). "Big Types in Little Runtime: Open-world Soundness and Collaborative Blame for Gradual Type Systems". In: *POPL 2017*. DOI: 10.1145/3009837.3009849.

Viviani, Giovanni (2022). "Leveraging developer discussions to improve design accessibility". PhD thesis. University of British Columbia. DOI: 10.14288/1.0421402.

Wadler, Philip (2015). "A Complement to Blame". In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. DOI: 10.4230/LIPIcs.SNAPL.2015.309.

Wadler, Philip and Robert Bruce Findler (2009). "Well-typed programs can't be blamed". In: *ESOP*. DOI: 10.1007/978-3-642-00590-9_1.

Wand, Mitchell (1986). "Finding the Source of Type Errors". In: *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '86, pp. 38–43. DOI: 10.1145/512644.512648.

Wand, Mitchell (1991). "Type inference for record concatenation and multiple inheritance". In: *Information and Computation* 93.1, pp. 1–15. DOI: 10.1016/0890-5401(91)90050-C.

Weiser, Mark (1981). "Program Slicing". In: ICSE '81. IEEE Press, pp. 439–449. URL: https://dl.acm.org/doi/10.5555/800078.802557.

Weiser, Mark (July 1982). "Programmers Use Slices When Debugging". In: *CACM* 25.7, pp. 446–452. DOI: 10.1145/358557.358577.

Williams, Jack, J. Garrett Morris, and Philip Wadler (Oct. 2018). "The Root Cause of Blame: Contracts for Intersection and Union Types". In: *Proc. ACM Program. Lang.* 2.OOPSLA. DOI: 10.1145/3276504.

Wise, Jenna, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine (Nov. 2020). "Gradual Verification of Recursive Heap Data Structures". In: *Proc. ACM Program. Lang.* 4.OOPSLA. DOI: 10.1145/3428296.

Wohlfahrt, Klaus (1962). "On Static and Dynamic Treatment of Types in Algol Translators". In: *Symbolic Languages in Data Processing: Proceedings of the Symposium organized*. International Computation Centre. New York: Gordon and Breach, pp. 325–330.

Woods, David D. (1984). "Visual momentum: a concept to improve the cognitive coupling of person and computer". In: *International Journal of Man-Machine Studies* 21.3, pp. 229–244. DOI: 10.1016/S0020-7373(84)80043-7.

Wrenn, John and Shriram Krishnamurthi (2017). "Error Messages Are Classifiers: A Process to Design and Evaluate Error Messages". In: Onward! '17. DOI: 10.1145/3133850.3133862.

Wright, Andrew K. and Matthias Felleisen (Nov. 1994). "A Syntactic Approach to Type Soundness". In: *Information and Computation* 115.1, pp. 38–94. DOI: 10.1006/inco.1994.1093.

Wrigstad, Tobias, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek (2010). "Integrating Typed and Untyped Code in a Scripting Language". In: POPL '10, pp. 377–388. DOI: 10.1145/1706299.1706343.

Wu, Yang, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo (2014). "Diagnosing Missing Events in Distributed Systems with Negative Provenance". In: *SIGCOMM*. DOI: 10.1145/2619239.2626335.

Ye, Wenjia, Matías Toro, and Federico Olmedo (Apr. 2023). "A Gradual Probabilistic Lambda Calculus". In: *Proc. ACM Program. Lang.* 7.OOPSLA1. DOI: 10.1145/3586036.

Yoo, James (2022). "Investigating data-flow reachability questions". MA thesis. University of British Columbia. DOI: 10.14288/1.0421073.

Zalewski, Jakub, James McKinna, J Garrett Morris, and Philip Wadler (2020). "λdB: Blame tracking at a higher fidelity". In: *WGT*. URL: https://wgt20.irif.fr/wgt20-final98-acmpaginated.pdf.

Zhang, Xiangyu, Neelam Gupta, and Rajiv Gupta (Apr. 2007). "A study of effectiveness of dynamic slicing in locating real faults". In: *Empirical Software Engineering* 12.2, pp. 143–160. DOI: 10.1007/s10664-006-9007-3.

Zuse, Konrad (Nov. 1948). "Über den Allgemeinen Plankalkül als Mittel zur Formulierung schematisch-kombinativer Aufgaben". In: *Archiv der Mathematik* 1.6, pp. 441–449. DOI: 10.1007/BF02038459.

A GTFL$_{\lesssim}$ OPERATOR DEFINITIONS

This section presents the full algorithmic definitions for some GTFL$_{\lesssim}$ operators that were omitted from the paper for lack of space.

Figure A.1 presents consistent subtype join and meet, which are used to define the type judgment for source and runtime programs.

Figure A.3 introduces an Inductive Definition of Initial Evidence, and Figure A.2 introduces an Inductive Definition of gradual meet. With these two, given Proposition 20, we can produce an inductive definition of consistent transitivity.

Figures A.4 and A.5 present a full direct equational definition of consistent transitivity for the original gradual rows-based representation. This definition was calculated from the AGT-based definition, then used to prove Proposition 20.

A.1 BOUNDED RECORDS AND BOUNDED ROWS

A.1.1 *Proofs of Galois Connection*

Lemma 8 (α is Sound)**.** *If $\widehat{T}$ is not empty, then $\widehat{T} \subseteq \gamma\left(\alpha\left(\widehat{T}\right)\right)$.*

Proof. By mutual induction on the structure of sets induced by the definition of the abstraction functions α and α^M.

$\boxed{\alpha \text{ cases}}$

- $\alpha\left(\mathscr{C}\left[\!\left[\mathsf{Int}\right]\!\right]\right) = \mathsf{Int}$.

  It is the case that $\mathscr{C}\left[\!\left[\mathsf{Int}\right]\!\right] \subseteq \{\,\mathsf{Int}\,\}$

- $\alpha\left(\mathscr{C}\left[\!\left[\mathsf{Bool}\right]\!\right]\right) = \mathsf{Bool}$.

  It is the case that $\mathscr{C}\left[\!\left[\mathsf{Bool}\right]\!\right] \subseteq \{\,\mathsf{Bool}\,\}$

- $\alpha\left(\mathscr{C}\left[\!\left[C_1 \to C_2\right]\!\right]\right) = \alpha\left(C_1\right) \to \alpha\left(C_2\right)$.

  Let $T_a \to T_b \in \mathscr{C}\left[\!\left[C_1 \to C_2\right]\!\right]$. By Induction Hypothesis, $T_a \in \gamma(\alpha(C_1))$ and $T_b \in \gamma(\alpha(C_2))$. By the definition of the concretization function, then $T_a \to T_b \in \gamma(\alpha(\mathscr{C}\left[\!\left[C_1 \to C_2\right]\!\right]))$.

- $\alpha\left(\mathscr{C}\left[\!\left[\sum_{i=1}^{n} \ell_i : {C^{\varnothing}}_i\right]\!\right]\right) = \left[\sum_{i=1}^{n} \ell_i : \alpha^M({C^{\varnothing}}_i)\right]$.

  Let $\left[\sum_{j=1}^{m} \ell_j : T_j\right] \in \mathscr{C}\left[\!\left[\sum_{i=1}^{n} \ell_i : {C^{\varnothing}}_i\right]\!\right]$.

  By mutual induction hypothesis on each label,
  $\left[\sum_{j=1}^{m} \ell_j : T_j\right] \in \mathscr{C}\left[\!\left[\sum_{i=1}^{n} \ell_i : \gamma^M(\alpha^M({C^{\varnothing}}_i))\right]\!\right]$. The labels missing in j from i must come from a set ${C^{\varnothing}}_i$ such that $\varnothing \in {C^{\varnothing}}_i$, and for all $C^{\varnothing}$, $\varnothing \in C^{\varnothing} \Rightarrow \varnothing \in \gamma^M(\alpha^M(C^{\varnothing}))$.

$\boxed{S \mathbin{\widetilde{\vee}} S}$ **Consistent Subtype Join**

$\widetilde{\vee} : \text{GType} \times \text{GType} \rightharpoonup \text{GType}$

$S_1 \mathbin{\widetilde{\vee}} S_2 = S_2 \mathbin{\widetilde{\vee}} S_1$

$? \mathbin{\widetilde{\vee}} ? = ?$

$\mathsf{Int} \mathbin{\widetilde{\vee}} \mathsf{Int} = \mathsf{Int}$

$\mathsf{Int} \mathbin{\widetilde{\vee}} ? = \mathsf{Int}$

$\mathsf{Bool} \mathbin{\widetilde{\vee}} \mathsf{Bool} = \mathsf{Bool}$

$\mathsf{Bool} \mathbin{\widetilde{\vee}} ? = \mathsf{Bool}$

$$(S_{11} \to S_{12}) \mathbin{\widetilde{\vee}} (S_{21} \to S_{22}) = (S_{11} \mathbin{\widetilde{\wedge}} S_{21}) \to (S_{12} \mathbin{\widetilde{\vee}} S_{22})$$

$(S_{11} \to S_{12}) \mathbin{\widetilde{\vee}} ? = (S_{11} \to S_{12}) \mathbin{\widetilde{\vee}} (? \to ?)$

$[\overline{l_i : S_{i1}}, *] \mathbin{\widetilde{\vee}} ? = [\overline{l_i : S_{i1}}, *] \mathbin{\widetilde{\vee}} [?]$

$$[\overline{l_i : S_{i1}}, \overline{l_j : S_j}] \mathbin{\widetilde{\vee}} [\overline{l_i : S_{i2}}, \overline{l_k : S_k}] = [\overline{l_i : S_{i1} \mathbin{\widetilde{\vee}} S_{i2}}]$$

$[\overline{l_i : S_{i1}}, *] \mathbin{\widetilde{\vee}} [\overline{l_i : S_{i2}}, \overline{l_k : S_k}, ?] = [\overline{l_i : S_{i1} \mathbin{\widetilde{\vee}} S_{i2}}, *]$

$$[\overline{l_i : S_{i1}}, \overline{l_j : S_j}^{+}, *] \mathbin{\widetilde{\vee}} [\overline{l_i : S_{i2}}, \overline{l_k : S_k}, ?] = [\overline{l_i : S_{i1} \mathbin{\widetilde{\vee}} S_{i2}}, ?]$$

$S \mathbin{\widetilde{\vee}} S$ undefined otherwise

$\boxed{S \mathbin{\widetilde{\wedge}} S}$ **Consistent Subtype Meet**

$\widetilde{\wedge} : \text{GType} \times \text{GType} \rightharpoonup \text{GType}$

$S_1 \mathbin{\widetilde{\wedge}} S_2 = S_2 \mathbin{\widetilde{\wedge}} S_1$

$? \mathbin{\widetilde{\wedge}} ? = ?$

$\mathsf{Int} \mathbin{\widetilde{\wedge}} \mathsf{Int} = \mathsf{Int}$

$\mathsf{Int} \mathbin{\widetilde{\wedge}} ? = \mathsf{Int}$

$\mathsf{Bool} \mathbin{\widetilde{\wedge}} \mathsf{Bool} = \mathsf{Bool}$

$\mathsf{Bool} \mathbin{\widetilde{\wedge}} ? = \mathsf{Bool}$

$$(S_{11} \to S_{12}) \mathbin{\widetilde{\wedge}} (S_{21} \to S_{22}) = (S_{11} \mathbin{\widetilde{\vee}} S_{21}) \to (S_{12} \mathbin{\widetilde{\wedge}} S_{22})$$

$(S_{11} \to S_{12}) \mathbin{\widetilde{\wedge}} ? = (S_{11} \to S_{12}) \mathbin{\widetilde{\wedge}} (? \to ?)$

$[\overline{l_i : S_{i1}}, *] \mathbin{\widetilde{\wedge}} ? = [\overline{l_i : S_{i1}}, *] \mathbin{\widetilde{\wedge}} [?]$

$$[\overline{l_i : S_{i1}}, \overline{l_j : S_j}] \mathbin{\widetilde{\wedge}} [\overline{l_i : S_{i2}}, \overline{l_k : S_k}] = [\overline{l_i : S_{i1} \mathbin{\widetilde{\wedge}} S_{i2}}, \overline{l_j : S_j}, \overline{l_k : S_k}]$$

$$[\overline{l_i : S_{i1}}, \overline{l_j : S_j}, ?] \mathbin{\widetilde{\wedge}} [\overline{l_i : S_{i2}}, \overline{l_k : S_k}] = [\overline{l_i : S_{i1} \mathbin{\widetilde{\wedge}} S_{i2}}, \overline{l_j : S_j}, \overline{l_k : S_k \mathbin{\widetilde{\wedge}} ?}, ?]$$

$$[\overline{l_i : S_{i1}}, \overline{l_j : S_j}, ?] \mathbin{\widetilde{\wedge}} [\overline{l_i : S_{i2}}, \overline{l_k : S_k}, ?] = [\overline{l_i : S_{i1} \mathbin{\widetilde{\wedge}} S_{i2}}, \overline{l_j : S_j \mathbin{\widetilde{\wedge}} ?}, \overline{l_k : S_k \mathbin{\widetilde{\wedge}} ?}, ?]$$

$S \mathbin{\widetilde{\wedge}} S$ undefined otherwise

Figure A.1: $\text{GTFL}_{\widetilde{<}}$: Consistent Subtype Extrema

$\boxed{S \sqcap S}$ **Gradual Meet**

$$
\begin{aligned}
S_1 \sqcap S_2 &= S_2 \sqcap S_1 \\
? \sqcap ? &= ? \\
\mathsf{Int} \sqcap \mathsf{Int} &= \mathsf{Int} \\
\mathsf{Bool} \sqcap \mathsf{Bool} &= \mathsf{Bool} \\
S \sqcap ? &= S \\
(S_{11} \to S_{12}) \sqcap (S_{21} \to S_{22}) &= (S_{11} \sqcap S_{21}) \to (S_{12} \sqcap S_{22}) \\
[\overline{l_i : S_{i1}}, *_1] \sqcap [\overline{l_i : S_{i2}}, *_2] &= [\overline{l_i : S_{i1} \sqcap S_{i1}}, *_1 \sqcap *_2] \\
&\qquad \text{where } *_1 \sqcap *_2 = \begin{cases} ? & \langle *_1, *_2 \rangle = \langle ?, ? \rangle \\ \emptyset & \text{otherwise} \end{cases} \\
[\overline{l_i : S_{i1}}, \overline{l_j : S_j}^{+}, *_1] \sqcap [\overline{l_i : S_{i2}}, ?] &= [\overline{l_i : S_{i1}}, \overline{l_j : S_j}^{+}, *_1] \sqcap [\overline{l_i : S_{i2}}, \overline{l_j : ?}^{+}, ?] \\
[\overline{l_i : S_{i1}}, \overline{l_j : S_j}^{+}, ?] \sqcap [\overline{l_i : S_{i2}}, \overline{l_k : S_k}^{+}, ?] &= \\
&\quad [\overline{l_i : S_{i1}}, \overline{l_j : S_j}^{+}, \overline{l_k : ?}^{+}, ?] \sqcap [\overline{l_i : S_{i2}}, \overline{l_j : ?}^{+}, \overline{l_k : S_k}^{+}, ?] \\
S_1 \sqcap S_2 &\text{ undefined otherwise}
\end{aligned}
$$

Figure A.2: Gradual Meet

$\boxed{\mathcal{I}[\![S \lesssim S]\!] : \textsc{GType} \times \textsc{GType} \rightharpoonup \textsc{Ev}}$ **Initial Evidence**

$$
\begin{aligned}
\mathcal{I}[\![S \lesssim S]\!] &= \langle S, S \rangle \quad \text{where } S \in \{\, \mathsf{Int}, \mathsf{Bool}, ? \,\} \\
\mathcal{I}[\![S \lesssim ?]\!] &= \langle S, S \rangle \quad \text{where } S \in \{\, \mathsf{Int}, \mathsf{Bool}, [*] \,\} \\
\mathcal{I}[\![? \lesssim S]\!] &= \langle S, S \rangle \quad \text{where } S \in \{\, \mathsf{Int}, \mathsf{Bool} \,\} \\
\mathcal{I}[\![S_{11} \to S_{12} \lesssim ?]\!] &= \mathcal{I}[\![S_{11} \to S_{12} \lesssim ? \to ?]\!] \\
\mathcal{I}[\![? \lesssim S_{21} \to S_{22}]\!] &= \mathcal{I}[\![? \to ? \lesssim S_{21} \to S_{22}]\!] \\
\mathcal{I}[\![S_{11} \to S_{12} \lesssim S_{21} \to S_{22}]\!] &= \langle S'_{11} \to S'_{12}, S'_{21} \to S'_{22} \rangle \\
&\qquad \text{where } \mathcal{I}[\![S_{21} \lesssim S_{11}]\!] = \langle S'_{21}, S'_{11} \rangle \\
&\qquad \text{and } \mathcal{I}[\![S_{12} \lesssim S_{22}]\!] = \langle S'_{12}, S'_{22} \rangle \\
\mathcal{I}[\![? \lesssim [\overline{l_i : S_i}, *]]\!] &= \mathcal{I}[\![[?] \lesssim [\overline{l_i : S_i}, *]]\!] \\
\mathcal{I}[\![[\overline{l_i : S_i}^{+}, *] \lesssim ?]\!] &= \langle [\overline{l_i : S_i}^{+}, *], [?] \rangle \\
&\quad \text{In the next 3 definitions, assume } \overline{\mathcal{I}[\![S_{i1} \lesssim S_{i2}]\!] = \langle S'_{i1}, S'_{i2} \rangle} \\
\mathcal{I}[\![[\overline{l_i : S_{i1}}] \lesssim [\overline{l_i : S_{i2}}, *]]\!] &= \langle [\overline{l_i : S'_{i1}}], [\overline{l_i : S'_{i2}}] \rangle \\
\mathcal{I}[\![[\overline{l_i : S_{i1}}, ?] \lesssim [\overline{l_i : S_{i2}}, *]]\!] &= \langle [\overline{l_i : S'_{i1}}, ?], [\overline{l_i : S'_{i2}}, *] \rangle \\
\mathcal{I}[\![[\overline{l_i : S_{i1}}, \overline{l_j : S_j}^{+}, *_1] \lesssim [\overline{l_i : S_{i2}}, *_2]]\!] &= \langle [\overline{l_i : S'_{i1}}, \overline{l_j : S_j}^{+}, *_1], [\overline{l_i : S'_{i2}}, *_2] \rangle \\
\mathcal{I}[\![[\overline{l_i : S_{i1}}, \overline{l_j : S_j}, ?] \lesssim [\overline{l_i : S_{i2}}, \overline{l_k : S_k}^{+}, *]]\!] &= \\
&\quad \mathcal{I}[\![[\overline{l_i : S_{i1}}, \overline{l_j : S_j}, \overline{l_k : ?}^{+}, ?] \lesssim [\overline{l_i : S_{i2}}, \overline{l_k : S_k}^{+}, *]]\!] \\
\mathcal{I}[\![S_1 \lesssim S_2]\!] &\text{ undefined otherwise}
\end{aligned}
$$

Figure A.3: $\text{GTFL}_{\lesssim}$: Definition of Initial Evidence

- $\alpha\left(\mathscr{C}\left[\!\left[? \sum_{i=1}^{n} \ell_i : C^{\varnothing}{}_i\right]\!\right]\right) = \left[\sum_{i=1}^{n} \ell_i : \alpha^M(C^{\varnothing}{}_i)\ ?\right]$.

  Let $\left[\sum_{j=1}^{m} \ell_j : T_j\right] \in \mathscr{C}\left[\!\left[? \sum_{i=1}^{n} \ell_i : C^{\varnothing}{}_i\right]\!\right]$. Therefore, $\sum_{j=1}^{m} \ell_j : T_j = \sum_{k=1}^{p} \ell_p : T_p \sum_{h=1}^{q} \ell_q : T_q$, where $\left[\sum_{k=1}^{p} \ell_k : T_k\right] \in \mathscr{C}\left[\!\left[\sum_{i=1}^{n} \ell_i : C^{\varnothing}{}_i\right]\!\right]$ and $\left[\sum_{h=1}^{q} \ell_h : T_h\right] \in \mathscr{C}\left[\!\left[\sum_{h=1}^{q} \ell_h : \mathcal{P}(\textsc{Type} \cup \{\varnothing\})\right]\!\right]$, where $dom(\ell_i) \cap dom(\ell_h) = \emptyset$. We follow the mutual induction hypothesis on each restricted label ℓ_k as in the previous case to know $\left[\sum_{k=1}^{p} \ell_k : T_k\right] \in \mathscr{C}\left[\!\left[\sum_{i=1}^{n} \ell_i : \gamma^M(\alpha^M(C^{\varnothing}{}_i))\right]\!\right]$, and since the abstraction generates a bounded row, we can extract from its concretization the appropriate type for the rest of the mappings.

- $\alpha\,(C) = ?$.

  Trivial since $\gamma\,(\alpha(C)) = \textsc{Type}$.

$\boxed{\alpha^M \text{ cases}}$

- $\alpha^M\,(\{\varnothing\}) = \varnothing$. It is the case that $\{\varnothing\} \subseteq \{\varnothing\}$.

- $\alpha^M\,(C) = (\alpha(C))_R$, $\varnothing \notin C$.

  Let $T \in C$. By mutual induction hypothesis, $T \in \gamma(\alpha(C))$.

- $\alpha^M\,(\{\varnothing\} \cup C) = (\alpha(C)_O)$.

  If C is empty, we fall back to the previous case. If C is not empty, we case by the kind of elements of the set:

  - $\varnothing \in \{\varnothing\} \cup C$. By definition of $\gamma^M(S_O)$, $\varnothing \in \gamma(\alpha(C)_O)$.
  - $T \in C$. By mutual induction hypothesis, $T \in \gamma(\alpha(C))$.

□

Lemma 9 (α is Optimal)**.** *If $\widehat{T}$ is not empty and $\widehat{T} \subseteq \gamma\,(S)$, then $\alpha\left(\widehat{T}\right) \sqsubseteq S$*

Proof. Remember that $S_1 \sqsubseteq S_2$ iff $\gamma(S_1) \subseteq \gamma(S_2)$. Then we proceed by mutual structural induction on the definitions of S and M.

$\boxed{S \text{ cases}}$

- $S = \mathsf{Int}$.

  Only nonempty subset of $\gamma(S)$ is $\{\mathsf{Int}\}$, and $\alpha(\{\mathsf{Int}\}) = \mathsf{Int}$.

- $S = \mathsf{Bool}$.

  Only nonempty subset of $\gamma(S)$ is $\{\mathsf{Bool}\}$, and $\alpha(\{\mathsf{Bool}\}) = \mathsf{Bool}$.

- $S = S_1 \to S_2$. By induction hypothesis.

- $S = ?$

  Trivial since $\gamma(?) = \textsc{Type}$.

- $S = \left[\sum_{i=1}^{n} \ell_i : M_i\right]$ Since $\widehat{T} \subseteq \gamma(S)$, $\alpha(\widehat{T}) = \left[\sum_{j=1}^{m} \ell_j : M_j\right]$, where $dom(\ell_j) \subseteq dom(\ell_i)$.

  Let $T = [\overline{l_a : T_a}] \in \gamma(\alpha(\widehat{T}))$. The label domain for T must be a subset of the domain of S and for every mapping $\ell_a : T_a$, by definition and induction hypothesis, $T_a \in \gamma^M(M_j)$ for some j such that $\ell_a = \ell_j$.

- $S = \left[\sum_{i=1}^{n} \ell_i : M_i \ ?\right]$

  Depending on the subset taken from the concretization, abstraction might generate a bounded row or a bounded record. In any case, all types in the concretization of that type would be a subset of those in the original row.

  The interesting case is when we get a smaller set that still generates a row. For any record type in this concretization, there is a potential set labels for which their type came from the row designator ?, but those will all have been in the original concretization of the row. For those mappings in the type that now come from declared labels in the bounded row, by induction hypothesis, their types will be in the concretization. We do not need to worry about the set of absent labels, as by shrinking the set on the left, the set of absent labels can only grow, which will only impact the concretization by making it smaller, as we want to.

$\boxed{M \text{ cases}}$

- $M = \{\varnothing\}$.

  Only nonempty subset of $\gamma^M(M)$ is $\{\varnothing\}$, and $\alpha^M(\{\varnothing\}) = \varnothing$.

- $M = S_R$.

  Since $\varnothing$ is not a member of $\gamma^M(S_R)$, we can appeal to the mutual induction hypothesis directly.

- $M = S_O$.

  Let's consider separately the case we take a subset containing $\varnothing$ or not.

  - If $\varnothing$ not in the set, then we can appeal to the mutual induction hypothesis directly.
  - If $\varnothing$ in the set, we must check for the rest of the set contents. If the set is the singleton $\{\varnothing\}$, $\gamma^M(S_O)$ contains it. If there is any other elements, we remove $\varnothing$ for the set and appeal to the mutual induction hypothesis, while later adding $\varnothing$ on both sides of the inequality.

□

Theorem 43 (Bounded Rows form a Galois Connection)**.**

Proof. By Sound and Optimality Lemmas. □

A.1.2 *BRR is Gamma-Complete*

In this section we prove the property that BRR is gamma-complete.

Lemma 10 (Consistent transitivity preserves evidence well-formedness)**.** *For every two evidence objects* $\vdash \varepsilon_1$ **wf** *and* $\vdash \varepsilon_2$ **wf**, *if there exists* $\varepsilon_3 = \varepsilon_1 \fatsemi^{<:} \varepsilon_2$, *then* $\vdash \varepsilon_3$ **wf**.

Proof. By structural induction over the definition of the well-formedness judgment. The only interesting cases are those with Bounded Records/Rows in both evidence objects, and the definition of $\fatsemi^{<:}$ in terms of $\gamma^{<:}(\varepsilon_1) \,;\, \gamma^{<:}(\varepsilon_2)$ guarantees that a well-formed evidence object is produced. □

Theorem 44 (Gamma-completeness of BRR)**.** *For every two evidence objects* $\vdash \varepsilon_1$ **wf** *and* $\vdash \varepsilon_2$ **wf**,

$$\gamma^{<:}(\varepsilon_1) \,;\, \gamma^{<:}(\varepsilon_2) = \gamma^{<:}(\varepsilon_1 \fatsemi^{<:} \varepsilon_2)$$

Proof. By definition of evidence composition, $\gamma^{<:}(\varepsilon_1 \fatsemi^{<:} \varepsilon_2) = \gamma^{<:}(\alpha^{<:}(\gamma^{<:}(\varepsilon_1) \,;\, \gamma^{<:}(\varepsilon_2)))$.

By Soundness of the Galois Connection, $\gamma^{<:}(\varepsilon_1 \fatsemi^{<:} \varepsilon_2) \supseteq \gamma^{<:}(\varepsilon_1) \,;\, \gamma^{<:}(\varepsilon_2)$, thus we are only left to prove set containment on the other direction.

Now, by Lemma 2, if $\gamma^{<:}(\varepsilon_1 \fatsemi^{<:} \varepsilon_2)$ is defined, there exists ε_3 such that $\gamma^{<:}(\varepsilon_1) \,;\, \gamma^{<:}(\varepsilon_2) = \gamma^{<:}(\varepsilon_3)$, and thus by Optimality of the galois connection, $\gamma^{<:}(\varepsilon_1 \fatsemi^{<:} \varepsilon_2) \subseteq (\gamma^{<:}(\varepsilon_1) \,;\, \gamma^{<:}(\varepsilon_2))$. □

A.1.3 *Definition of Initial Evidence*

We present Initial evidence for BRR as if Bounded Records and Rows were available to the programmer in Figure A.6 . This presentation is useful in writing an inductive definition of evidence composition, and does not impact the definition of initial evidence with BRRs for GTFL$_{\lesssim}$, as we can apply the same rules for Gradual Rows and Record, under the condition for building initial evidence that Gradual Records become Bounded Records where every mapping in the Gradual Record is included and annotated as Required, and Gradual Rows become Bounded Rows where every mapping in the Gradual Row is included and annotated and Required.

In combination with the definition of gradual meet for BRR presented in Figure A.7, we can build an algorithmic definition of evidence composition.

A.1.4 *Definition of Consistent Subtype Meet and Join for BRR*

The original statically typed language we are basing our systems upon includes conditional branching via the if construct. To assign appropriate types in the context of branching, the static system includes the Meet ($\wedge$) and Join ($\vee$) operations traversing the subtyping lattice over static types defined in Figure A.8:

We include the gradual versions of these definitions in the context of BRR in Figures A.9 and A.10.

$\boxed{\fatsemi : \mathrm{Ev} \times \mathrm{Ev} \rightharpoonup \mathrm{Ev}}$ **Consistent Transitivity**

$$
\begin{aligned}
\langle ?, ? \rangle \fatsemi \langle ?, ? \rangle &= \langle ?, ? \rangle \\
\langle S, S \rangle \fatsemi \langle ?, ? \rangle &= \langle S, S \rangle \qquad \text{where } S \in \{ \mathsf{Int}, \mathsf{Bool} \} \\
\langle ?, ? \rangle \fatsemi \langle S, S \rangle &= \langle S, S \rangle \qquad \text{where } S \in \{ \mathsf{Int}, \mathsf{Bool} \} \\
\langle S_{11} \to S_{12}, S_{21} \to S_{22} \rangle \fatsemi \langle ?, ? \rangle &= \langle S_{11} \to S_{12}, S_{21} \to S_{22} \rangle \fatsemi \langle ? \to ?, ? \to ? \rangle \\
\langle ?, ? \rangle \fatsemi \langle S_{11} \to S_{12}, S_{21} \to S_{22} \rangle &= \langle ? \to ?, ? \to ? \rangle \fatsemi \langle S_{11} \to S_{12}, S_{21} \to S_{22} \rangle \\
\langle ?, ? \rangle \fatsemi \langle [\overline{l_i : S_i}, *_1], [\overline{l_j : S_j}, *_2] \rangle &= \langle [?], [?] \rangle \fatsemi \langle [\overline{l_i : S_i}, *_1], [\overline{l_j : S_j}, *_2] \rangle \\
\langle [\overline{l_i : S_i}, *_1], [\overline{l_j : S_j}, *_2] \rangle \fatsemi \langle ?, ? \rangle &= \langle [\overline{l_i : S_i}, *_1], [\overline{l_j : S_j}, *_2] \rangle \fatsemi \langle [?], [?] \rangle
\end{aligned}
$$

$$
\begin{aligned}
&\langle S, S \rangle \fatsemi \langle S, S \rangle = \langle S, S \rangle \qquad \text{where } S \in \{ \mathsf{Int}, \mathsf{Bool} \} \\
&\langle S_{11} \to S_{12}, S_{21} \to S_{22} \rangle \fatsemi \langle S_{31} \to S_{32}, S_{41} \to S_{42} \rangle = \langle S_{51} \to S_{52}, S_{61} \to S_{62} \rangle \\
&\qquad \text{where } \langle S_{41}, S_{31} \rangle \fatsemi \langle S_{21}, S_{11} \rangle = \langle S_{61}, S_{51} \rangle, \quad \langle S_{12}, S_{22} \rangle \fatsemi \langle S_{32}, S_{42} \rangle = \langle S_{52}, S_{62} \rangle
\end{aligned}
$$

$$
\langle [\overline{l_i : S_{i1}}, *_1], [\overline{l_i : S_{i2}}, *_2] \rangle \fatsemi \langle [\overline{l_i : S_{i3}}, *_3], [\overline{l_i : S_{i4}}, *_4] \rangle = \langle [\overline{l_i : S_{i5}}, *_5], [\overline{l_i : S_{i6}}, *_6] \rangle
$$

where $\overline{\langle S_{i1}, S_{i2} \rangle \fatsemi \langle S_{i3}, S_{i4} \rangle = \langle S_{i5}, S_{i6} \rangle}$,

$\langle *_1, *_2 \rangle$	$\langle *_3, *_4 \rangle$	$= \langle *_5, *_6 \rangle$
$\langle \emptyset, \emptyset \rangle$	$\langle *_3, *_4 \rangle$	$\langle \emptyset, \emptyset \rangle$
$\langle ?, ? \rangle$	$\langle ?, ? \rangle$	$\langle ?, ? \rangle$
else		$\langle ?, \emptyset \rangle$

$$
\begin{aligned}
&\langle [\overline{l_i : S_{i1}}, \overline{l_j : S_{j1}}^{+}, *_1], [\overline{l_i : S_{i2}}, \overline{l_j : S_{j2}}^{+}, *_2] \rangle \fatsemi \langle [\overline{l_i : S_{i3}}, ?], [\overline{l_i : S_{i4}}, *_4] \rangle = \\
&\qquad \langle [\overline{l_i : S_{i1}}, \overline{l_j : S_{j1}}^{+}, *_1], [\overline{l_i : S_{i2}}, \overline{l_j : S_{j2}}^{+}, *_2] \rangle \fatsemi \langle [\overline{l_i : S_{i3}}, \overline{l_j : ?}^{+}, ?], [\overline{l_i : S_{i4}}, *_4] \rangle
\end{aligned}
$$

$$
\begin{aligned}
&\langle [\overline{l_i : S_{i1}}, ?], [\overline{l_i : S_{i2}}, ?] \rangle \fatsemi \langle [\overline{l_i : S_{i3}}, \overline{l_k : S_{k3}}^{+}, *_3], [\overline{l_i : S_{i4}}, \overline{l_k : S_{k4}}^{+}, *_4] \rangle = \\
&\qquad \langle [\overline{l_i : S_{i1}}, \overline{l_k : ?}^{+}, ?], [\overline{l_i : S_{i2}}, \overline{l_k : ?}^{+}, ?] \rangle \fatsemi \langle [\overline{l_i : S_{i3}}, \overline{l_k : S_{k3}}^{+}, *_3], [\overline{l_i : S_{i4}}, \overline{l_k : S_{k4}}^{+} *_4] \rangle
\end{aligned}
$$

$$
\begin{aligned}
&\langle [\overline{l_i : S_{i1}}, \overline{l_j : S_{j1}}^{+}, ?], [\overline{l_i : S_{i2}}, \overline{l_j : S_{j2}}^{+}, ?] \rangle \fatsemi \langle [\overline{l_i : S_{i3}}, \overline{l_k : S_{k3}}^{+}, ?], [\overline{l_i : S_{i4}}, \overline{l_k : S_{k4}}^{+}, *_4] \rangle = \\
&\qquad \langle [\overline{l_i : S_{i1}}, \overline{l_j : S_{j1}}^{+}, \overline{l_k : ?}^{+}, ?], [\overline{l_i : S_{i2}}, \overline{l_j : S_{j2}}^{+}, \overline{l_k : ?}^{+}, ?] \rangle \\
&\qquad\qquad \fatsemi \langle [\overline{l_i : S_{i3}}, \overline{l_k : S_{k3}}, \overline{l_j : ?}^{+}, ?], [\overline{l_i : S_{i4}}, \overline{l_k : S_{k4}}, *_4] \rangle
\end{aligned}
$$

Figure A.4: Consistent Transitivity: Part 1

$\boxed{\fatsemi : \mathrm{Ev} \times \mathrm{Ev} \rightharpoonup \mathrm{Ev}}$ **Consistent Transitivity (cont'd.)**

$$\langle [\overline{l_i : S_{i1}}, \overline{l_j : S_{j1}}^{\oplus_j}, \overline{l_k : S_k}^{\oplus_k}, *_1], [\overline{l_i : S_{i2}}, \overline{l_j : S_{j2}}^{\oplus_j}] \rangle \fatsemi \langle [\overline{l_i : S_{i3}}, \overline{l_j : S_{j3}}^{\oplus_j}], [\overline{l_i : S_{i4}}, *_4] \rangle = \langle [\overline{l_i : S_{i5}}, \overline{l_j : S_{j5}}^{\oplus_j}, \overline{l_k : S_k}^{\oplus_k}, *_1], [\overline{l_i : S_{i6}}, *_4] \rangle$$

$$\langle [\overline{l_i : S_{i1}}, \overline{l_j : S_{j1}}^{\oplus_j}, \overline{l_q : S_{q1}}, \overline{l_k : S_k}^{\oplus_k}, *_1], [\overline{l_i : S_{i2}}, \overline{l_j : S_{j2}}^{\oplus_j}, \overline{l_q : S_{q2}}] \rangle \fatsemi \langle [\overline{l_i : S_{i3}}, \overline{l_j : S_{j3}}^{\oplus_j}, ?], [\overline{l_i : S_{i4}}, *_4] \rangle = \langle [\overline{l_i : S_{i5}}, \overline{l_j : S_{j5}}^{\oplus_j}, \overline{l_q : S_{q1}}, \overline{l_k : S_k}^{\oplus_k}, *_1], [\overline{l_i : S_{i6}}, *_4] \rangle$$

$$\langle [\overline{l_i : S_{i1}}, \overline{l_m : S_{m1}}, \overline{l_j : S_{j1}}^{\oplus_j}, \overline{l_n : S_{n1}}^{\oplus_n}, \overline{l_k : S_k}^{\oplus_k}, *_1], [\overline{l_i : S_{i2}}, \overline{l_j : S_{j2}}^{\oplus_j}, ?] \rangle \fatsemi \langle [\overline{l_i : S_{i3}}, \overline{l_m : S_{m3}}, \overline{l_p : S_{p3}}, \overline{l_j : S_{j3}}^{\oplus_j}, \overline{l_n : S_{n3}}^{\oplus_n}, \overline{l_r : S_{r3}}^{\oplus_r}, *_3], [\overline{l_i : S_{i4}}, \overline{l_m : S_{m4}}, \overline{l_p : S_{p4}}, *_4] \rangle = \langle [\overline{l_i : S_{i5}}, \overline{l_m : S_{m5}}, \overline{l_p : S_{p5}}, \overline{l_j : S_{j5}}^{\oplus_j}, \overline{l_n : S_{n5}}^{\oplus_n}, \overline{l_r : S_{r5}}^{\oplus_r}, \overline{l_k : S_k}^{\oplus_k}, *_1], [\overline{l_i : S_{i6}}, \overline{l_m : S_{m6}}, \overline{l_p : S_{p6}}, *_4] \rangle$$

$*_1 = ?$ if $\{ \overline{l_p}, \overline{l_r}^{\oplus_r} \} \neq \emptyset$

$$\langle [\overline{l_i : S_{i1}}, \overline{l_m : S_{m1}}, \overline{l_j : S_{j1}}^{\oplus_j}, \overline{l_n : S_{n1}}^{\oplus_n}, \overline{l_q : S_{q1}}^{+}, \overline{l_k : S_k}^{\oplus_k}, *_1], [\overline{l_i : S_{i2}}, \overline{l_j : S_{j2}}^{\oplus_j}, \overline{l_q : S_{q2}}^{+}, ?] \rangle \fatsemi$$

$$\langle [\overline{l_i : S_{i3}}, \overline{l_m : S_{m3}}, \overline{l_p : S_{p3}}, \overline{l_j : S_{j3}}^{\oplus_j}, \overline{l_n : S_{n3}}^{\oplus_n}, \overline{l_r : S_{r3}}^{\oplus_r}, ?], [\overline{l_i : S_{i4}}, \overline{l_m : S_{m4}}, \overline{l_p : S_{p4}}, *_4] \rangle$$

$$=$$

$$\langle [\overline{l_i : S_{i1}}, \overline{l_m : S_{m1}}, \overline{l_j : S_{j1}}^{\oplus_j}, \overline{l_n : S_{n1}}^{\oplus_n}, \overline{l_q : S_{q1}}^{+}, \overline{l_k : S_k}^{\oplus_k}, *_1], [\overline{l_i : S_{i2}}, \overline{l_j : S_{j2}}^{\oplus_j}, \overline{l_q : S_{q2}}^{+}, ?] \rangle \fatsemi \langle [\overline{l_i : S_{i3}}, \overline{l_m : S_{m3}}, \overline{l_p : S_{p3}}, \overline{l_j : S_{j3}}^{\oplus_j}, \overline{l_n : S_{n3}}^{\oplus_n}, \overline{l_r : S_{r3}}^{\oplus_r}, \overline{l_q : ?}^{+}, ?], [\overline{l_i : S_{i4}}, \overline{l_m : S_{m4}}, \overline{l_p : S_{p4}}, *_4] \rangle$$

where $\overline{\langle S_{i1}, S_{i2} \rangle \fatsemi \langle S_{i3}, S_{i4} \rangle = \langle S_{i5}, S_{i6} \rangle}$, $\overline{\langle S_{j1}, S_{j2} \rangle \fatsemi \langle S_{j3}, S_{j3} \rangle = \langle S_{j5}, S_{j6} \rangle}$

$\overline{\mathcal{I}[\![S_{m1} \lesssim ?]\!] \fatsemi \langle S_{m3}, S_{m4} \rangle = \langle S_{m5}, S_{m6} \rangle}$, $\overline{\mathcal{I}[\![S_{n1} \lesssim ?]\!] \fatsemi \langle S_{n3}, S_{n3} \rangle = \langle S_{n5}, S_{n6} \rangle}$

$\overline{\langle ?, ? \rangle \fatsemi \langle S_{p3}, S_{p4} \rangle = \langle S_{p5}, S_{p6} \rangle}$, $\overline{\langle ?, ? \rangle \fatsemi \langle S_{r3}, S_{r3} \rangle = \langle S_{r5}, S_{r6} \rangle}$

$$\langle \oplus_{j+n+r}, \oplus_k \rangle \in \{ \langle \emptyset, + \rangle, \langle +, \emptyset \rangle, \langle +, + \rangle \}, \oplus_{j+n+r} = \begin{cases} \oplus_j & \oplus_n, \oplus_r \text{ do not appear} \\ \emptyset & \langle \oplus_j, \oplus_n, \oplus_r \rangle = \langle \emptyset, \emptyset, \emptyset \rangle \\ + & \text{otherwise} \end{cases}$$

$\langle S_1, S_2 \rangle \fatsemi \langle S_3, S_4 \rangle$ undefined otherwise

Figure A.5: Consistent Transitivity: Part 2

$\boxed{\mathcal{I}[\![S \lesssim S]\!] : \textsc{GType} \times \textsc{GType} \rightharpoonup \textsc{Ev}}$

$$
\begin{array}{rll}
\mathcal{I}[\![S \lesssim S]\!] & = \langle S, S\rangle & S \in \{\,\mathsf{Int}, \mathsf{Bool}, ?\,\} \\
\mathcal{I}[\![S \lesssim ?]\!] & = \langle S, S\rangle & S \in \{\,\mathsf{Int}, \mathsf{Bool}\,\} \\
\mathcal{I}[\![? \lesssim S]\!] & = \langle S, S\rangle & S \in \{\,\mathsf{Int}, \mathsf{Bool}\,\} \\
\mathcal{I}[\![S_{11} \to S_{12} \lesssim ?]\!] & = \mathcal{I}[\![S_{11} \to S_{12} \lesssim ? \to ?]\!] & \\
\mathcal{I}[\![? \lesssim S_{21} \to S_{22}]\!] & = \mathcal{I}[\![? \to ? \lesssim S_{21} \to S_{22}]\!] & \\
\mathcal{I}[\![S_{11} \to S_{12} \lesssim S_{21} \to S_{22}]\!] & = \langle S'_{11} \to S'_{12}, S'_{21} \to S'_{22}\rangle & \\
 & \quad \text{where } \mathcal{I}[\![S_{21} \lesssim S_{11}]\!] = \langle S'_{21}, S'_{11}\rangle & \\
 & \qquad\quad \mathcal{I}[\![S_{12} \lesssim S_{22}]\!] = \langle S'_{12}, S'_{22}\rangle & \\
\mathcal{I}[\![? \lesssim \left[\sum_{i=1}^{n} \ell_i : M_i *_2\right]]\!] & = \mathcal{I}[\![[?] \lesssim \left[\sum_{i=1}^{n} \ell_i : M_i *_2\right]]\!] & \\
\mathcal{I}[\![\left[\sum_{i=1}^{n} \ell_i : M_i *_1\right] \lesssim ?]\!] & = \mathcal{I}[\![\left[\sum_{i=1}^{n} \ell_i : M_i *_1\right] \lesssim [?]]\!] & \\
\mathcal{I}[\![\left[\sum_{i=1}^{n} \ell_i : M_{i1} \sum_{j=1}^{m} \ell_j : M_j *_1\right] \lesssim \left[\sum_{i=1}^{n} \ell_i : M_{i2} \sum_{k=1}^{p} \ell_k : M_k *_2\right]]\!] & = \left\langle \left[\sum_{i=1}^{n} \ell_i : M'_{i1} \sum_{j=1}^{m} \ell_j : M'_{j1} \sum_{k=1}^{p} \ell_k : M'_{k1} *_1\right], \left[\sum_{i=1}^{n} \ell_i : M'_{i2} \sum_{j=1}^{m} \ell_j : M'_{j2} \sum_{k=1}^{p} \ell_k : M'_{k2} *_4\right]\right\rangle & \\
 & \quad \text{where} & \\
 & \quad \text{for every } i, \mathcal{I}[\![M_{i1} \lesssim M_{i2}]\!] = \langle M'_{i1}, M'_{i2}\rangle & \\
 & \quad \text{for every } j, \mathcal{I}[\![M_j \lesssim D(*_2)]\!] = \langle M'_{j1}, M'_{j2}\rangle & \\
 & \quad \text{for every } k, \mathcal{I}[\![D(*_1) \lesssim M_k]\!] = \langle M'_{k1}, M'_{k2}\rangle & \\
 & \quad *_4 = \begin{cases} ? & \text{if } *_1 = *_2 = ? \\ \cdot & \text{otherwise} \end{cases} & \\
 & \quad D(\cdot) = \varnothing & \\
 & \quad D(?) = ?_O &
\end{array}
$$

$\boxed{\mathcal{I}[\![M \lesssim M]\!] : \textsc{Mapping} \times \textsc{Mapping} \rightharpoonup \textsc{Mapping} * \textsc{Mapping}}$

$$
\begin{array}{rcl}
\mathcal{I}[\![M \lesssim \varnothing]\!] & = & \langle M, \varnothing\rangle \\
\mathcal{I}[\![(S_1)_{*1} \lesssim (S_2)_R]\!] & = & \langle (S'_1)_R, (S'_2)_R\rangle \\
 & & \quad \text{where } \mathcal{I}[\![S_1 \lesssim S_2]\!] = \langle S'_1, S'_2\rangle \\
\mathcal{I}[\![(S_1)_{*1} \lesssim (S_2)_O]\!] & = & \begin{cases} \langle\, S_1 \,_{*1}, (S'_2)_O\rangle & \text{where } \mathcal{I}[\![S_1 \lesssim S_2]\!] = \langle S'_1, S'_2\rangle \\ \langle\, S_1 \,_{*1}, \varnothing\rangle & \text{if } \mathcal{I}[\![S_1 \lesssim S_2]\!] \text{ is undefined.} \end{cases}
\end{array}
$$

Figure A.6: Definition of Initial Evidence for Bounded Rows

$\boxed{S \sqcap S}$ **Gradual Meet for BRR**

$$
\begin{aligned}
S_1 \sqcap S_2 &= S_2 \sqcap S_1 \\
? \sqcap ? &= ? \\
\mathsf{Int} \sqcap \mathsf{Int} &= \mathsf{Int} \\
\mathsf{Bool} \sqcap \mathsf{Bool} &= \mathsf{Bool} \\
S \sqcap ? &= S \\
(S_{11} \to S_{12}) \sqcap (S_{21} \to S_{22}) &= (S_{11} \sqcap S_{21}) \to (S_{12} \sqcap S_{22}) \\
\left[\sum_{i=1}^{n} \ell_i : M_{i1} \sum_{j=1}^{m} \ell_j : M_j *_1\right] &= \left[\sum_{i=1}^{n} \ell_i : M_{i1} \sqcap M_{i2} \sum_{j=1}^{m} \ell_j : M_j \sqcap D(*_2) \sum_{k=1}^{p} \ell_k : D(*_1) \sqcap M_k *_3\right] \\
&\sqcap \\
\left[\sum_{i=1}^{n} \ell_i : M_{i2} \sum_{k=1}^{p} \ell_k : M_k *_2\right] &
\end{aligned}
$$

$$
\text{where } *_3 = \begin{cases} ? & \text{if } *_1 = *_2 = ? \\ \cdot & \text{otherwise} \end{cases}
$$

$$
S_1 \sqcap S_2 \text{ undefined otherwise}
$$

$\boxed{M \sqcap M}$ **Gradual Meet for BRR Mappings**

$$
\begin{aligned}
M_1 \sqcap M_2 &= M_2 \sqcap M_1 \\
\varnothing \sqcap \varnothing &= \varnothing \\
\varnothing \sqcap S_O &= \varnothing \\
(S_1)_R \sqcap (S_2)_* &= (S_1 \sqcap S_2)_R \\
(S_1)_O \sqcap (S_2)_O &= \begin{cases} (S_1 \sqcap S_2)_O & \text{if } S_1 \sqcap S_2 \text{ defined} \\ \varnothing & \text{if } S_1 \sqcap S_2 \text{ undefined} \end{cases}
\end{aligned}
$$

Figure A.7: Gradual Meet for BRR

$\boxed{T \ddot{\vee} T}$ **Static Subtype Join**

$\ddot{\vee} : \textsc{Type} \times \textsc{Type} \rightharpoonup \textsc{Type}$

$T_1 \ddot{\vee} T_2 = T_2 \ddot{\vee} T_1$

$\mathsf{Int} \ddot{\vee} \mathsf{Int} = \mathsf{Int}$

$\mathsf{Bool} \ddot{\vee} \mathsf{Bool} = \mathsf{Bool}$

$$(T_{11} \to T_{12}) \ddot{\vee} (T_{21} \to T_{22}) = (T_{11} \underset{\cdot\cdot}{\wedge} T_{21}) \to (T_{12} \ddot{\vee} T_{22})$$

$$\left[\sum_{i=1}^{n} \ell_i : T_{i1} \sum_{j=1}^{m} \ell_j : T_j\right] \ddot{\vee} \left[\sum_{i=1}^{n} \ell_i : T_{i2} \sum_{k=1}^{p} \ell_k : T_k\right] = \left[\sum_{i=1}^{n} \ell_i : T_{i1} \ddot{\vee} T_{i2}\right]$$

$\boxed{T \underset{\cdot\cdot}{\wedge} T}$ **Static Subtype Meet**

$\underset{\cdot\cdot}{\wedge} : \textsc{Type} \times \textsc{Type} \rightharpoonup \textsc{Type}$

$T_1 \underset{\cdot\cdot}{\wedge} T_2 = T_2 \underset{\cdot\cdot}{\wedge} T_1$

$\mathsf{Int} \underset{\cdot\cdot}{\wedge} \mathsf{Int} = \mathsf{Int}$

$\mathsf{Bool} \underset{\cdot\cdot}{\wedge} \mathsf{Bool} = \mathsf{Bool}$

$$(T_{11} \to T_{12}) \underset{\cdot\cdot}{\wedge} (T_{21} \to T_{22}) = (T_{11} \ddot{\vee} T_{21}) \to (T_{12} \underset{\cdot\cdot}{\wedge} T_{22})$$

$$\left[\sum_{i=1}^{n} \ell_i : T_{i1} \sum_{j=1}^{m} \ell_j : T_j\right] \underset{\cdot\cdot}{\wedge} \left[\sum_{i=1}^{n} \ell_i : T_{i2} \sum_{k=1}^{p} \ell_k : T_k\right] = \left[\sum_{i=1}^{n} \ell_i : T_{i1} \underset{\cdot\cdot}{\wedge} T_{i2} \sum_{j=1}^{m} \ell_j : T_j \sum_{k=1}^{p} \ell_k : T_k\right]$$

Figure A.8: Static Subtype Meet and Join

$\boxed{S \mathbin{\widetilde{\vee}} S}$ **Consistent Subtype Join with BRR**

$\widetilde{\vee} : \textsc{GType} \times \textsc{GType} \rightharpoonup \textsc{GType}$

$S_1 \mathbin{\widetilde{\vee}} S_2 = S_2 \mathbin{\widetilde{\vee}} S_1$

$? \mathbin{\widetilde{\vee}} ? = ?$

$\mathsf{Int} \mathbin{\widetilde{\vee}} \mathsf{Int} = \mathsf{Int}$

$\mathsf{Int} \mathbin{\widetilde{\vee}} ? = \mathsf{Int}$

$\mathsf{Bool} \mathbin{\widetilde{\vee}} \mathsf{Bool} = \mathsf{Bool}$

$\mathsf{Bool} \mathbin{\widetilde{\vee}} ? = \mathsf{Bool}$

$$(S_{11} \to S_{12}) \mathbin{\widetilde{\vee}} (S_{21} \to S_{22}) = \\ (S_{11} \mathbin{\widetilde{\wedge}} S_{21}) \to (S_{12} \mathbin{\widetilde{\vee}} S_{22})$$

$(S_{11} \to S_{12}) \mathbin{\widetilde{\vee}} ? = (S_{11} \to S_{12}) \mathbin{\widetilde{\vee}} (? \to ?)$

$$\left[\sum_{i=1}^{n} \ell_i : M_i *\right] \mathbin{\widetilde{\vee}} ? = \left[\sum_{i=1}^{n} \ell_i : M_i *\right] \mathbin{\widetilde{\vee}} [?]$$

$$\left[\sum_{i=1}^{n} \ell_i : M_{i1} \sum_{j=1}^{m} \ell_j : M_j *_1\right] \mathbin{\widetilde{\vee}} \left[\sum_{i=1}^{n} \ell_i : M_{i2} \sum_{k=1}^{p} \ell_k : M_k *_2\right] = \\ \left[\sum_{i=1}^{n} \ell_i : M_{i1} \mathbin{\widetilde{\vee}} M_{i2} \sum_{j=1}^{m} \ell_j : M_j \mathbin{\widetilde{\vee}} D(*_2) \sum_{k=1}^{p} \ell_k : D(*_1) \mathbin{\widetilde{\vee}} M_k *_3\right]$$

$$\text{where } *_3 = \begin{cases} ? & \text{if } *_1 = *_2 = ? \\ \cdot & \text{otherwise} \end{cases}$$

$S \mathbin{\widetilde{\vee}} S$ undefined otherwise

$\boxed{M \mathbin{\widetilde{\vee}} M}$ **Consistent Subtype Join with BRR**

$\widetilde{\vee} : \textsc{Mapping} \times \textsc{Mapping} \rightharpoonup \textsc{Mapping}$

$M_1 \mathbin{\widetilde{\vee}} M_2 = M_2 \mathbin{\widetilde{\vee}} M_1$

$\varnothing \mathbin{\widetilde{\vee}} M = \varnothing$

$(S_1)_R \mathbin{\widetilde{\vee}} (S_2)_R = (S_1 \mathbin{\widetilde{\vee}} S_2)_R$

$$(S_1)_O \mathbin{\widetilde{\vee}} (S_2)_* = \begin{cases} (S_1 \mathbin{\widetilde{\vee}} S_2)_O & \text{if } S_1 \mathbin{\widetilde{\vee}} S_2 \text{ defined} \\ \varnothing & \text{if } S_1 \mathbin{\widetilde{\vee}} S_2 \text{ undefined} \end{cases}$$

Figure A.9: Consistent Subtype Extrema with BRR, part 1

$\boxed{S \mathbin{\widetilde{\wedge}} S}$ **Consistent Subtype Meet with BRR**

$\widetilde{\wedge} : \textsc{GType} \times \textsc{GType} \rightharpoonup \textsc{GType}$

$S_1 \mathbin{\widetilde{\wedge}} S_2 = S_2 \mathbin{\widetilde{\wedge}} S_1$

$? \mathbin{\widetilde{\wedge}} ? = ?$

$\mathsf{Int} \mathbin{\widetilde{\wedge}} \mathsf{Int} = \mathsf{Int}$

$\mathsf{Int} \mathbin{\widetilde{\wedge}} ? = \mathsf{Int}$

$\mathsf{Bool} \mathbin{\widetilde{\wedge}} \mathsf{Bool} = \mathsf{Bool}$

$\mathsf{Bool} \mathbin{\widetilde{\wedge}} ? = \mathsf{Bool}$

$$(S_{11} \to S_{12}) \mathbin{\widetilde{\wedge}} (S_{21} \to S_{22}) = \\ (S_{11} \mathbin{\widetilde{\vee}} S_{21}) \to (S_{12} \mathbin{\widetilde{\wedge}} S_{22})$$

$(S_{11} \to S_{12}) \mathbin{\widetilde{\wedge}} ? = (S_{11} \to S_{12}) \mathbin{\widetilde{\wedge}} (? \to ?)$

$$\left[\sum_{i=1}^{n} \ell_i : M_i *\right] \mathbin{\widetilde{\wedge}} ? = \left[\sum_{i=1}^{n} \ell_i : M_i *\right] \mathbin{\widetilde{\wedge}} [?]$$

$$\left[\sum_{i=1}^{n} \ell_i : M_{i1} \sum_{j=1}^{m} \ell_j : M_j *_1\right] \mathbin{\widetilde{\wedge}} \left[\sum_{i=1}^{n} \ell_i : M_{i2} \sum_{k=1}^{p} \ell_k : M_k *_2\right] = \\ \left[\sum_{i=1}^{n} \ell_i : M_{i1} \mathbin{\widetilde{\wedge}} M_{i2} \sum_{j=1}^{m} \ell_j : M_j \mathbin{\widetilde{\wedge}} D(*_2) \sum_{k=1}^{p} \ell_k : D(*_1) \mathbin{\widetilde{\wedge}} M_k *_3\right]$$

$$\text{where } *_3 = \begin{cases} \cdot & \text{if } *_1 = *_2 = \cdot \\ ? & \text{otherwise} \end{cases}$$

$S \mathbin{\widetilde{\wedge}} S$ undefined otherwise

$\boxed{M \mathbin{\widetilde{\wedge}} M}$ **Consistent Subtype Meet with BRR**

$\widetilde{\wedge} : \textsc{Mapping} \times \textsc{Mapping} \rightharpoonup \textsc{Mapping}$

$M_1 \mathbin{\widetilde{\wedge}} M_2 = M_2 \mathbin{\widetilde{\wedge}} M_1$

$\varnothing \mathbin{\widetilde{\wedge}} M = M$

$(S_1)_R \mathbin{\widetilde{\wedge}} (S_2)_* = (S_1 \mathbin{\widetilde{\wedge}} S_2)_R$

$$(S_1)_O \mathbin{\widetilde{\wedge}} (S_2)_O = \begin{cases} (S_1 \mathbin{\widetilde{\wedge}} S_2)_O & \text{if } S_1 \mathbin{\widetilde{\wedge}} S_2 \text{ defined} \\ \varnothing & \text{if } S_1 \mathbin{\widetilde{\wedge}} S_2 \text{ undefined} \end{cases}$$

Figure A.10: Consistent Subtype Extrema with BRR, Part 2

B AGTLC ERR-APPROXIMATES NATURAL

Lemma 11 (From Evidence Composition to Casts)**.** *Let $\varepsilon_1;\varepsilon_2 = \varepsilon_3$ such that $\varepsilon_1 \vdash S_1 \sim S_2$, $\varepsilon_2 \vdash S_2 \sim S_3$, and $\varepsilon_3 \vdash S_1 \sim S_3$. For any well-typed values u of type S_1, there exists a reduction in the natural semantics such that $\langle S_3 \Leftarrow S_2\rangle\langle S_2 \Leftarrow S_1\rangle u \longrightarrow^* v$*

Proof. By mutual induction on gradual types S_1, S_2, and S_3 (to deal with contravariance on function type domains), and cases on $\varepsilon_1;\varepsilon_2 = \varepsilon_3$.

- Case $S_1 = \mathsf{Int}$. Because composition succeeds, both S_2 and S_3 are either Int or ?. There are notions of reduction for either case.
- Case $S_1 = ?$ impossible, no raw values of this type.
- Case $S_1 = S_1' \to S_2'$. Because composition succeeds, both S_2 and S_3 are either ? or functions. In 1 or 2 steps using the notions of reduction, we obtain a new function value of the expected type that contains nested applications.

□

Lemma 12 (Simulation for evidence composition with values)**.** *Let $\varepsilon_1;\varepsilon_2 = \varepsilon_3$ such that $\varepsilon_1 \vdash S_1 \sim S_2$, $\varepsilon_2 \vdash S_2 \sim S_3$, and $\varepsilon_3 \vdash S_1 \sim S_3$. For any well-typed values u and $\mathbf{u}$ such that $\mathbf{e} \approx u : S_1$, and $\langle S_3 \Leftarrow S_2\rangle\langle S_2 \Leftarrow S_1\rangle u \longrightarrow^* v$, it is the case that $\varepsilon_3\mathbf{u} \approx v : S_3$.*

Proof. By cases on the derivation of $\mathbf{u} \approx u : S_1$, which can only happen by rules [S-C] or [S-E]. We can then use either rules [S-F], [S-G] or [S-D] to build a derivation such that $\varepsilon_3\mathbf{u} \approx v : S_3$. □

Lemma 13 (Substitution preserves simulation)**.** *Let $\mathbf{e} \approx e : S$ and $\mathbf{v} \approx v : S'$. Then $[\mathbf{v}/\mathbf{x}]_{S'}\mathbf{e} \approx [v/x]_{S'}e : S$.*

Proof. By structural induction on the derivation of $\mathbf{e} \approx e : S$. □

Lemma 14 (Reduction preserves simulation (if))**.** *Let $\mathbf{e} \sim e : S$. If $\mathbf{e} \longrightarrow \mathbf{e}'$, then exists some $n \geq 1$ such that after n steps for e ($e \longrightarrow_n e'$), $\mathbf{e}' \sim e' : S$.*

Proof. By structural induction on the definition of the simulation relation, then by cases on the applicable reduction rules.

Rules [S-A], [S-B], [S-C], [S-E], [S-F] and [S-G] do never step.

For the other rules, each rule can be distinguished into two possible cases: Whether the reduction happens immediately at the top level (base case), or if the reduction happens by induction on the structure of $\mathbf{e}$. When the reduction happens inductively, the induction hypothesis can be applied, to obtain a multi-step natural reduction. This reduction can be altered at each step to extend the evaluation context with the additions

Syntax

$$x \in \textsc{Var}, \quad n \in \mathbb{Z}, \quad \Gamma \in \textsc{Env} = \textsc{Var} \xrightarrow{\text{fin}} \textsc{Type}$$

$S \in \textsc{GType}$	$::=$	$\mathsf{Int} \mid S \to S \mid ?$	(gradual types)
$t \in \textsc{Term}$	$::=$	$n \mid x \mid \lambda\,(x : S)\,.\,t \mid t\ t \mid \langle S \Leftarrow S \rangle t$	(terms)
$u \in \textsc{RawValue}$	$::=$	$n \mid \lambda\,(x : S)\,.\,t$	(raw values)
$v \in \textsc{Value}$	$::=$	$u \mid (\langle ? \Leftarrow S \rangle u)$	(values)
$E \in \textsc{ECtxt}$	$::=$	$\square \mid E[F[\square]]$	(evaluation contexts)
$F \in \textsc{EvFrame}$	$::=$	$\square\ t \mid v\ \square \mid \langle S \Leftarrow S \rangle \square$	(evaluation frames)

$\boxed{t \rightsquigarrow t}$ **Notions of Reduction**

$$(\lambda\,(x : S)\,.\,e)\ \ v \rightsquigarrow [v/x]\,e$$
$$\langle \mathsf{Int} \Leftarrow \mathsf{Int} \rangle u \rightsquigarrow u$$
$$\langle S_1 \to S_2 \Leftarrow S_3 \to S_4 \rangle u \rightsquigarrow (\lambda(x : S_3)\,.\,\langle S_4 \Leftarrow S_2 \rangle (u\ \langle S_3 \Leftarrow S_1 \rangle x))$$
$$\langle ? \Leftarrow ? \rangle \langle ? \Leftarrow S \rangle u \rightsquigarrow \langle ? \Leftarrow S \rangle u$$
$$\langle \mathsf{Int} \Leftarrow ? \rangle \langle ? \Leftarrow \mathsf{Int} \rangle u \rightsquigarrow u$$
$$\langle S_1 \to S_2 \Leftarrow ? \rangle \langle ? \Leftarrow S_3 \to S_4 \rangle u \rightsquigarrow \langle S_1 \to S_2 \Leftarrow S_3 \to S_4 \rangle u$$
$$\langle \mathsf{Int} \Leftarrow ? \rangle \langle ? \Leftarrow S_1 \to S_2 \rangle u \rightsquigarrow \textbf{error}$$
$$\langle S_1 \to S_2 \Leftarrow ? \rangle \langle ? \Leftarrow \mathsf{Int} \rangle u \rightsquigarrow \textbf{error}$$

$\boxed{t \longrightarrow t}$ **Contextual Reduction**

$$\text{Structural}\ \frac{e \rightsquigarrow e'}{E[e] \longrightarrow E[e']}$$

Figure B.1: Natural Small-step Semantics

$\boxed{\varepsilon \vdash S_1 \sim S_2}$

$$\frac{}{\varepsilon \vdash ? \sim ?} \qquad \frac{}{\langle \mathsf{Int}, \mathsf{Int} \rangle \vdash \mathsf{Int} \sim \mathsf{Int}} \qquad \frac{}{\langle \mathsf{Int}, \mathsf{Int} \rangle \vdash \mathsf{Int} \sim ?}$$

$$\frac{}{\langle \mathsf{Int}, \mathsf{Int} \rangle \vdash ? \sim \mathsf{Int}} \qquad \frac{\varepsilon_1 \vdash S_3 \sim S_1 \quad \varepsilon_2 \vdash S_2 \sim S_4}{\varepsilon_1 \to \varepsilon_2 \vdash S_1 \to S_2 \sim S_3 \to S_4}$$

$$\frac{\varepsilon_1 \vdash ? \sim S_1 \quad \varepsilon_2 \vdash S_2 \sim ?}{\varepsilon_1 \to \varepsilon_2 \vdash S_1 \to S_2 \sim ?} \qquad \frac{\varepsilon_1 \vdash S_3 \sim ? \quad \varepsilon_2 \vdash ? \sim S_4}{\varepsilon_1 \to \varepsilon_2 \vdash ? \sim S_3 \to S_4}$$

Figure B.2: Evidence that supports type consistency

$$\boxed{[\![\mathbf{e} \blacktriangleright \Gamma \vdash t : S]\!]_{\text{Natural}} = e_{\text{Natural}}}$$

$$\text{N-A}\ \frac{}{[\![\mathbf{n} \blacktriangleright \Gamma \vdash n : \mathsf{Int}]\!]_{\text{Natural}} = n} \qquad \text{N-B}\ \frac{[\![\mathbf{e} \blacktriangleright \Gamma \vdash t : S]\!]_{\text{Natural}} = t' \qquad \varepsilon \vdash S \sim S'}{[\![\varepsilon\mathbf{e} \blacktriangleright \Gamma \vdash (t :: S') : S']\!]_{\text{Natural}} = \langle S' \Leftarrow S \rangle t'}$$

$$\text{N-C}\ \frac{\overline{[\![\mathbf{e}_i \blacktriangleright \Gamma \vdash t_i : S_i]\!]_{\text{Natural}} = t'_i} \quad \varepsilon_1 \vdash S_1 \sim S'_1 \to S'_2 \qquad \varepsilon_2 \vdash S_2 \sim S'_1}{[\![(\varepsilon_1\mathbf{e}_1\ \varepsilon_2\mathbf{e}_2) \blacktriangleright \Gamma \vdash t_1\ t_2 : S'_2]\!]_{\text{Natural}} = (\langle S'_1 \to S'_2 \rangle t'_1)\ (\langle S'_1 \Leftarrow S_2 \rangle t'_2)}$$

$$\text{N-D}\ \frac{x : S' \in \Gamma}{[\![\mathbf{x} \blacktriangleright \Gamma \vdash x : S']\!]_{\text{Natural}} = x}$$

$$\text{N-E}\ \frac{[\![\mathbf{e} \blacktriangleright \Gamma, x : S_1 \vdash t : S_2]\!]_{\text{Natural}} = t'}{[\![(\lambda x.\mathbf{e}) \blacktriangleright \Gamma \vdash (\lambda x : S_1.t) : S_1 \to S_2]\!]_{\text{Natural}} = \lambda(x : S_1)\,.\,t'}$$

Figure B.3: Transformation from AGTLC into a Natural Semantics.

$$\boxed{(\mathbf{e} \text{ or } \mathbf{error}) \approx e_{\text{Natural}} : S}$$

$$\text{S-A}\ \frac{}{\mathbf{error} \approx e : S} \qquad \text{S-B}\ \frac{}{\mathbf{x} \approx x : S} \qquad \text{S-C}\ \frac{}{\mathbf{n} \approx n : \mathsf{Int}}$$

$$\text{S-D}\ \frac{\mathbf{e} \approx e : S_1 \qquad \varepsilon \vdash S_1 \sim S_2}{\varepsilon\mathbf{e} \approx \langle S_2 \Leftarrow S_1 \rangle e : S_2} \qquad \text{S-E}\ \frac{\mathbf{e} \approx e : S_2}{\lambda x\,.\,\mathbf{e} \approx \lambda(y : S_1)\,.\,e : S_1 \to S_2}$$

$$\text{S-F}\ \frac{\varepsilon \vdash S_1 \to S_2 \sim S_3 \to S_4 \qquad \mathbf{e} \approx e : S_2}{\varepsilon\lambda x\,.\,\mathbf{e} \approx (\lambda(x : S_3)\,.\,\langle S_4 \Leftarrow S_3 \rangle\,(\lambda(x : S_1)\,.\,e)\ \langle S_1 \Leftarrow S_3 \rangle x) : S_3 \to S_4}$$

$$\text{S-G}\ \frac{\varepsilon \vdash \mathsf{Int} \sim \mathsf{Int}}{\varepsilon\mathbf{n} \approx n : \mathsf{Int}}$$

$$\text{S-F}\ \frac{\mathbf{e}_1 \approx e_1 : S_1 \qquad \mathbf{e}_2 \approx e_2 : S_2 \qquad \varepsilon_1 \vdash S_1 \sim S'_1 \to S'_2 \qquad \varepsilon_2 \vdash S_2 \sim S'_1}{\varepsilon_1\mathbf{e}_1\ \varepsilon_2\mathbf{e}_2 \approx \langle S'_1 \to S'_2 \Leftarrow S_1 \rangle e_1\ \langle S'_1 \Leftarrow S_2 \rangle e_2}$$

Figure B.4: Simulation relation between ARL and Natural

on the conclusion of [S-D] and [S-F]. The key steps to address are instead when reduction happens immediately at the top level of **e**.

For rule [S-D], the case of interest is when $\mathbf{e} = \varepsilon_1\varepsilon_2\mathbf{u}$ and it steps to a new ascription $\varepsilon_3\mathbf{u}$, with $\varepsilon_1 \vdash S_1 \sim S_2$, $\varepsilon_2 \vdash S_2 \sim S_3$, and $\varepsilon_3 \vdash S_1 \sim S_3$. By Lemma 11, the natural cast composition reduces to a value. We can use then Lemma 12 to build the derivation needed.

For Rule [S-F], the case of interest is when the derivation has the following structure inductive structure:

$$\frac{\begin{array}{c}\lambda x\,.\,\mathbf{e} \approx \lambda(x:S_1)\,.\,e : S_1 \to S_3 \qquad \mathbf{u} \approx u : S_2 \\ \varepsilon_1 \vdash S_1 \to S_3 \sim S_1' \to S_2' \qquad \varepsilon_2 \vdash S_2 \sim S_1'\end{array}}{\varepsilon_1\lambda x\,.\,\mathbf{e}\ \varepsilon_2\mathbf{u} \approx (\langle S_1' \to S_2' \Leftarrow S_1 \to S_3\rangle\lambda(x:S_1)\,.\,e)\ \langle S_1' \Leftarrow S_2\rangle u : S_2'}$$

The ARL term beta-reduces to $icod(\varepsilon_1)\ ([(\varepsilon_2; idom(\varepsilon_1)\mathbf{u})/\mathbf{x}]\mathbf{e})$ The term on the right has two casts, and it takes the following steps of reduction

$$\begin{array}{rl}(\langle S_1' \to S_2' \Leftarrow S_1 \to S_3\rangle\lambda(x:S_1)\,.\,e)\ \langle S_1' \Leftarrow S_2\rangle u & \longrightarrow \\ (\lambda(x:S_1')\,.\,\langle S_2' \Leftarrow S_3\rangle((\lambda(x:S_1)\,.\,e)\ \langle S_1 \Leftarrow S_1'\rangle x))\ \langle S_1' \Leftarrow S_2\rangle u & \longrightarrow \\ \langle S_2' \Leftarrow S_3\rangle((\lambda(x:S_1)\,.\,e)\ \langle S_1 \Leftarrow S_1'\rangle\langle S_1' \Leftarrow S_2\rangle u & \end{array}$$

By Lemma 12 and Lemma 11 we know that $\langle S_1 \Leftarrow S_1'\rangle\langle S_1' \Leftarrow S_2\rangle u \longrightarrow^* v$ for some v and that $\varepsilon_2; idom(\varepsilon_1)\mathbf{u} \approx v : S_1$. We can perform then a beta-reduction step on the natural semantics.

From Lemma 13, we can prove that $icod(\varepsilon_1)\ ([(\varepsilon_2; idom(\varepsilon_1)\mathbf{u})/\mathbf{x}]\mathbf{e}) \approx \langle S_2' \Leftarrow S_3\rangle[v/x]e$. □

Lemma 15 (Reduction preserves simulation (iff))**.** *Let* $\mathbf{e} \sim e : S$. *If* $e \longrightarrow e'$, *there exists* e'' *such that* $e' \longrightarrow^* e''$ *and either* $\mathbf{e} \longrightarrow \mathbf{error}$ *or there exists an* $\mathbf{e}'$ *such that* $\mathbf{e} \longrightarrow \mathbf{e}'$ *and* $\mathbf{e}' \sim e'' : S$.

Note that if $\mathbf{e} \longrightarrow \mathbf{error}$, *then also* $\mathbf{error} \approx e'' : S$.

Proof. Because e takes a step, it is not a value. Because $\mathbf{e} \approx e : S$, **e** is also not a value. It can either take a step $\mathbf{e} \longrightarrow \mathbf{error}$ or $\mathbf{e} \longrightarrow \mathbf{e}'$. If $\mathbf{e} \longrightarrow \mathbf{error}$, the proof is trivial because of rule [S-A]. If there is an $\mathbf{e} \longrightarrow \mathbf{e}'$, we can follow for the proof the same inductive structure as the proof of Lemma 14, but instead of generating the $n \geq 1$ steps on the right, we complete the $n-1$ steps required to match the step $\mathbf{e}'$ and preserve the simulation. □

Lemma 16 (Transformation into Natural simulates)**.** *Whenever* $\mathbf{e} \blacktriangleright \Gamma \vdash t : S$, $rr \approx [\![\mathbf{e} \blacktriangleright \Gamma \vdash t : S]\!]_{Natural}$.

Proof. By structural induction on the derivation of runtime typing. The changes introduced by the transformation are consistent with the simulation relation. □

Theorem 45 (AGTLC Err-approximates Natural)**.** *For any program that* $\mathbf{e} \blacktriangleright \Gamma \vdash t : S$, $[\![\mathbf{e} \blacktriangleright \Gamma \vdash t : S]\!]_{Natural} \longrightarrow^*_{Natural} \mathbf{error}$ *implies that* $\mathbf{e} \longrightarrow^* \mathbf{error}$.

Proof. By Lemma 16, we know that $\mathbf{e} \approx [\![\mathbf{e} \blacktriangleright \Gamma \vdash t : S]\!]_{\text{Natural}} : S$ into the simulation. Note that $[\![\mathbf{e} \blacktriangleright \Gamma \vdash t : S]\!]_{\text{Natural}} \longrightarrow^* e \longrightarrow \mathbf{error}$. We can use Lemma 15 up to e, from which we

can either directly conclude that $\mathbf{e} \longrightarrow^* \mathbf{error}$ or reach related terms on the simulation ($\mathbf{e}' \approx e : S$).

If we reach related terms on the simulation, we are to prove still from $e \longrightarrow \mathbf{error}$ that $\mathbf{e}' \longrightarrow \mathbf{error}$. The cases that reduce an error in the natural semantics from e will have two incompatible casts, that is, $e = E[\langle S_3 \Leftarrow S_2 \rangle \langle S_2 \Leftarrow S_1 \rangle u]$ but $S_1 \not\sim S_3$.

These casts manifest in the simulation as having corresponding evidence objects in the ARL side. Because $S_1 \not\sim S_3$, the composition of the evidence objects supporting the judgements in the ARL term thus must fail, and the runtime term will step to error. $\square$

C GALOIS SLICING FOR NATURAL SEMANTICS AND FOR SMALL-STEP SEMANTICS

In this chapter I prove equivalences between different styles of slicing. I focus on forward slicing equivalences, as each forms their own Galois connection implying equivalences for Backward Slicing.

C.1 LANGUAGE WITH FUNCTIONS AND ADDITION

There is a multi-step approach for the proofs in the section. In the first step we go from natural environment-based semantics to natural substitution-based semantics, and in a second step we go from natural substitution-based semantics to small-step operational semantics.

For the first step, Figure C.1 introduces forward slicing using environments for a natural semantics. The rules in Figure C.2 use substitution instead of environments. Theorem 46 shows that there is an expression-equivalent (Definition 21) output slice with substitutions.

For the second step, Figure C.3 introduces forward slicing using a substitution-based small-step semantics. Theorem 47 shows that for any big-step substitution based forward slicing there exits a small-step multi-step forward slicing that produces the same output.

We can combine both steps to infer that for any big-step environment-based forward slicing, there exists a small-step multi-step forward slicing producing an expression-equivalent output to the big-step slicing.

First we need to define some auxiliary operations:

Definition 18 (Appling a substitution)**.** *We apply a substitution as follows:*

$$\frac{}{\underset{e[\cdot]=e}{\mathsf{subst}}\,(\mathring{e},\cdot) = \mathring{e}} \qquad \frac{\lfloor \mathring{v}, v \rfloor = (\mathring{e}_x, e_x) \qquad \underset{[e_x/x]e=e'}{[\mathring{e}_x/x]\mathring{e}} = \mathring{e}' \qquad \underset{e'[\rho]=e''}{\mathsf{subst}}\left(\mathring{e}', \mathring{rho}\right) = \mathring{e}''}{\underset{e[[x\mapsto v]\rho]=e''}{\mathsf{subst}}\,(\mathring{e}, [x \mapsto \mathring{v}]\mathring{\rho}) = \mathring{e}''}$$

This definition is mutually inductively defined with the following definition:

Definition 19 (Values to Expressions Conversion)**.** *We define the conversion from values to expressions as follows:*

$$\frac{\mathring{e} \sqsubseteq n}{\lfloor \mathring{e}, n \rfloor = (\mathring{e}, n)} \qquad \frac{\mathring{e} \sqsubseteq e \qquad \underset{e[\rho]=e'}{\mathsf{subst}}\,(\mathring{e}, \mathring{\rho}) = \mathring{e}'}{\lfloor \langle \mathring{\rho}, \mathsf{rec}\ f(x)\,.\,\mathring{e} \rangle, \langle \rho, \mathsf{rec}\ f(x)\,.\,e \rangle \rfloor = \left(\mathring{e}', e'\right)}$$

$$\frac{\underset{e[\rho]=e'}{\mathsf{subst}}\,(\blacksquare, \lfloor \rho \rfloor) = \blacksquare}{\lfloor \blacksquare, \langle \rho, \mathsf{rec}\ f(x)\,.\,e \rangle \rfloor = (\blacksquare, e')}$$

$$\boxed{\underset{\rho, e \Downarrow_\rho v}{\mathsf{fwd}}\ (\mathring{\rho}, \mathring{e}) = \mathring{v}}$$

$$\text{BE-A}\ \frac{}{\underset{\rho, x \Downarrow_\rho \rho(x)}{\mathsf{fwd}}\ (\mathring{\rho}, x) = \mathring{\rho}(x)} \qquad \text{BE-B}\ \frac{}{\underset{\rho, x \Downarrow_\rho \rho(x)}{\mathsf{fwd}}\ (\mathring{\rho}, \blacksquare) = \blacksquare} \qquad \text{BE-C}\ \frac{\mathring{e} \sqsubseteq n}{\underset{\rho, n \Downarrow_\rho n}{\mathsf{fwd}}\ (\mathring{\rho}, \mathring{e}) = \mathring{e}}$$

$$\text{BE-D}\ \frac{\underset{\rho, e_1 \Downarrow_\rho n_1}{\mathsf{fwd}}\ (\mathring{\rho}, \mathring{e}_1) = n_1 \qquad \underset{\rho, e_2 \Downarrow_\rho n_2}{\mathsf{fwd}}\ (\mathring{\rho}, \mathring{e}_2) = n_2}{\underset{\rho, e_1 + e_2 \Downarrow_\rho n_1 \hat{+} n_2}{\mathsf{fwd}}\ (\mathring{\rho}, \mathring{e}_1 + \mathring{e}_2) = n_1 \hat{+} n_2}$$

$$\text{BE-E}\ \frac{\underset{\rho, e_1 \Downarrow_\rho n_1}{\mathsf{fwd}}\ (\mathring{\rho}, \mathring{e}_1) = \mathring{e}_1' \qquad \underset{\rho, e_2 \Downarrow_\rho n_2}{\mathsf{fwd}}\ (\mathring{\rho}, \mathring{e}_2) = \mathring{e}_2' \qquad \mathring{e}_1' = \blacksquare \text{ or } \mathring{e}_2' = \blacksquare}{\underset{\rho, e_1 + e_2 \Downarrow_\rho n_1 \hat{+} n_2}{\mathsf{fwd}}\ (\mathring{\rho}, \mathring{e}_1 + \mathring{e}_2) = \blacksquare}$$

$$\text{BE-F}\ \frac{\underset{\rho, e_1 \Downarrow_\rho n_1}{\mathsf{fwd}}\ (\mathring{\rho}, \blacksquare) = \blacksquare \qquad \underset{\rho, e_2 \Downarrow_\rho n_2}{\mathsf{fwd}}\ (\mathring{\rho}, \blacksquare) = \blacksquare}{\underset{\rho, e_1 + e_2 \Downarrow_\rho n_1 \hat{+} n_2}{\mathsf{fwd}}\ (\mathring{\rho}, \blacksquare) = \blacksquare}$$

$$\text{BE-G}\ \frac{}{\underset{\rho, \mathsf{rec}\ f(x).e \Downarrow_\rho \langle \rho, \mathsf{rec}\ f(x).e \rangle}{\mathsf{fwd}}\ (\mathring{\rho}, \mathsf{rec}\ f(x)\,.\,\mathring{e}) = \langle \mathring{\rho}, \mathsf{rec}\ f(x)\,.\,\mathring{e} \rangle}$$

$$\text{BE-H}\ \frac{}{\underset{\rho, \mathsf{rec}\ f(x).e \Downarrow_\rho \langle \rho, \mathsf{rec}\ f(x).e \rangle}{\mathsf{fwd}}\ (\mathring{\rho}, \blacksquare) = \blacksquare}$$

$$\text{BE-I}\ \frac{\begin{array}{c}\underset{\rho, e_1 \Downarrow_\rho v_1}{\mathsf{fwd}}\ (\mathring{\rho}, \mathring{e}_1) = \mathring{v}_1 \qquad \underset{\rho, e_2 \Downarrow_\rho v_2}{\mathsf{fwd}}\ (\mathring{\rho}, \mathring{e}_2) = \mathring{v}_2 \qquad v_1 = \langle \rho'', \mathsf{rec}\ f(x)\,.\,e \rangle \\ \underset{[f \mapsto v_1][x \mapsto v_2]\rho'', e \Downarrow_\rho v}{\mathsf{fwd}}\ ([f \mapsto \mathring{v}_1][x \mapsto \mathring{v}_2]\mathring{\rho}'', \mathring{e}) = \mathring{v}\mathring{v}_1 = \langle \mathring{\rho}'', \mathsf{rec}\ f(x)\,.\,\mathring{e} \rangle\end{array}}{\underset{\rho, e_1\ e_2 \Downarrow_\rho v}{\mathsf{fwd}}\ (\mathring{\rho}, \mathring{e}_1\ \mathring{e}_2) = \mathring{v}}$$

$$\text{BE-J}\ \frac{\begin{array}{c}\underset{\rho, e_1 \Downarrow_\rho v_1}{\mathsf{fwd}}\ (\mathring{\rho}, \blacksquare) = \blacksquare \qquad \underset{\rho, e_2 \Downarrow_\rho v_2}{\mathsf{fwd}}\ (\mathring{\rho}, \mathring{e}_2) = \mathring{v}_2 \qquad v_1 = \langle \rho'', \mathsf{rec}\ f(x)\,.\,e \rangle \\ \underset{[f \mapsto v_1][x \mapsto v_2]\rho'', e \Downarrow_\rho v}{\mathsf{fwd}}\ \left([f \mapsto \blacksquare][x \mapsto \blacksquare]\mathring{\rho}'', \blacksquare\right) = \blacksquare\end{array}}{\underset{\rho, e_1\ e_2 \Downarrow_\rho v}{\mathsf{fwd}}\ (\mathring{\rho}, \blacksquare\ \mathring{e}_2) = \mathring{v}}$$

$$\text{BE-K}\ \frac{\begin{array}{c}\underset{\rho, e_1 \Downarrow_\rho v_1}{\mathsf{fwd}}\ (\mathring{\rho}, \blacksquare) = \blacksquare \qquad \underset{\rho, e_2 \Downarrow_\rho v_2}{\mathsf{fwd}}\ (\mathring{\rho}, \blacksquare) = \blacksquare \qquad v_1 = \langle \rho'', \mathsf{rec}\ f(x)\,.\,e \rangle \\ \underset{[f \mapsto v_1][x \mapsto v_2]\rho'', e \Downarrow_\rho v}{\mathsf{fwd}}\ \left([f \mapsto \blacksquare][x \mapsto \blacksquare]\mathring{\rho}'', \blacksquare\right) = \blacksquare\end{array}}{\underset{\rho, e_1\ e_2 \Downarrow_\rho v}{\mathsf{fwd}}\ (\mathring{\rho}, \blacksquare) = \mathring{v}}$$

Figure C.1: Big-step environment based semantics

As a convention, the function names and arguments cannot appear in the closure environment.

Definition 20 (Closed values)**.** *We define* $\mathsf{closed}(v)$ *as values that have no free variables (this definition is relevant only for closures in this language).*

Lemma 17 (Closed values are substitution-immune)**.** *Let* $\lfloor \mathring{v}, v \rfloor = (\mathring{e}, e)$ *and* $\mathsf{closed}(v)$*. For any environments,*

$$\underset{e[\rho]=e'}{\mathsf{subst}}\,(\mathring{e}, \mathring{\rho}) = \mathring{e}' \text{ iff } \underset{e[\rho']=e'}{\mathsf{subst}}\left(\mathring{e}, \mathring{\rho}'\right) = \mathring{e}'$$

Proof. By mutual structural induction on the definition of "values to expression conversion" and "applying a substitution" for the hypothesis. Because values are closed, any free variables have been replaced already by $\lfloor \mathring{v}, v \rfloor = (\mathring{e}, e)$. So the later substitutions act as the identity function. □

Definition 21 (Expression-equivalent values)**.** *I say two values* v *and* v' *(and their respective slices* $\mathring{v}$ *and* $\mathring{v}'$*) are expression-equivalent if their values to expression conversion produces the same terms. That is,*

$$\lfloor \mathring{v}, v \rfloor = (\mathring{e}_v, e_v) \text{ iff } \lfloor \mathring{v}', v' \rfloor = (\mathring{e}_v, e_v)$$

Lemma 18 (Closed values reduce to a expression-equivalent value)**.** *Let* $\lfloor \mathring{v}, v \rfloor = (\mathring{e}, e)$ *and* $\mathsf{closed}(v)$*,*

$$\text{if } \underset{\rho, e \Downarrow_\rho v'}{\mathsf{fwd}}\,(\mathring{\rho}, \mathring{e}) = \mathring{v}' \text{ then } \lfloor \mathring{v}', v' \rfloor = (\mathring{e}, e)$$

Proof. By cases. If $v = n$, trivial. If v is a closure, converting it to an expression will generate a closed function, which reduces to a new closure. However, this converting this new closure to an expression is a no-op as no free terms were left on the body of the function expression. □

Definition 22 (Environment split)**.** *We define* $\rho = \rho' +\!\!+ \rho''$ *as an* environment split*, which divides an environment into two domain-disjoint environments. By convention, we present environment splits as preserving the slices partial ordering (that is, when an environment is split, any slice of it must also be split, and the splits are assumed to properly split the domains).*

Lemma 19 (Expression-equivalent environment changes produce expression-equivalent values)**.** *Let* $\underset{\rho, e \Downarrow_\rho v}{\mathsf{fwd}}\,(\mathring{\rho}, \mathring{e}) = \mathring{v}$*, for any* $\mathring{\rho}' \sqsubseteq \rho'$ *such that the domains of* ρ *and* ρ' *are equivalent and their mappings (with their slices) are expression-equivalent, then there exists an equivalent trace that produces an output that is expression-equivalent to the output of the original trace, that is, there exists* $\underset{\rho', e \Downarrow_\rho v'}{\mathsf{fwd}}\,(\mathring{\rho}', \mathring{e}) = \mathring{v}'$ *and*

$$\lfloor \mathring{v}, v \rfloor = (\mathring{e}_v, e_v) \text{ iff } \lfloor \mathring{v}', v' \rfloor = (\mathring{e}_v, e_v)$$

Proof. By structural induction on the derivation of forward slicing for the hypothesis (following the rules from Figure C.1). At the key step, for rule [BE-I], applying the induction hypothesis on the first two premises provides expression-equivalent mappings

that can be used to build an expression-equivalent environment for input to applying the induction hypothesis to the third premise forward slicing premise. □

Lemma 20 (Environment substitution preserves an expression-equivalent output value)**.** *Let* $\underset{\rho_1 +\!\!+ \rho_2, e \Downarrow_\rho v}{\mathsf{fwd}} (\mathring{\rho}_1 +\!\!+ \mathring{\rho}_2, \mathring{e}) = \mathring{v}$ *and* $\mathsf{closed}(v)$.

When the substitution is partially applied, there is a big-step evaluation trace that preserves expression-equivalence of the outcoming values. That is,

$$\textit{for } \underset{e\lfloor\rho_1\rfloor = e'}{\mathsf{subst}} (\mathring{e}, \mathring{\rho}_1) = \mathring{e}' \textit{ and } \underset{\rho_2, e' \Downarrow_\rho v'}{\mathsf{fwd}} (\mathring{\rho}_2, \mathring{e}') = \mathring{v}' \textit{ then}$$

$$\lfloor \mathring{v}, v \rfloor = (\mathring{e}_v, e_v) \textit{ iff } \lfloor \mathring{v}', v' \rfloor = (\mathring{e}_v, e_v)$$

Proof. By structural induction on the rules from Figure C.1 on the hypothesis. The interesting rules are:

- Rule [BE-A]. By nested induction on ρ. When the substitution is performed, it follows by Lemma 18.
- Rule [BE-B]. Any substitution on ■ produces ■, and the rest is as for rule [BE-A].
- Rule [BE-I]. We can assume that neither f or x are in the domain of ρ_1 or $\mathring{\rho}_1$, as the restriction of $v_1 = \langle \rho'', \mathsf{rec}\ f(x) \,.\, e$, impedes substitution from replacing either identifier.

  By Lemma 17, we can apply induction on the first two forward slicing premises. The key extra step is relying upon Lemma 19 after applying the induction hypothesis on the third slicing premise.

□

Lemma 21 (Depth equivalence for induction purposes)**.** *Both forward slicing derivations in Lemma 20 have the same depth.*

Proof. By structural induction on the derivation of the first slicing derivation. The key step is reaching the leaves of the derivation that use either rules [BE-A] or [BE-B]. After substitution, every reference to [BE-B] becomes either a reference to either [BE-C] with $\mathring{e} =$ ■ or to [BE-H], which have the same size Every reference to [BE-A] becomes a reference to either [BE-C], [BE-G], or [BE-H]. They all have the same depth (1). □

Definition 23 (Expression-values)**.** *To avoid confusion between values (as in closures), and expression-like values, I introduce the following syntactic category for purposes of this appendix:*

Syntax

$$\begin{array}{rcll} u \in \textsc{ExprValue} & ::= & n \mid \mathsf{rec}\ f(x) \,.\, e & \textit{(expression-values)} \\ \mathring{u} \in \textsc{ExprValueSlice} & ::= & n \mid \mathsf{rec}\ f(x) \,.\, \mathring{e} \mid \blacksquare & \textit{(expression-value slices)} \end{array}$$

This syntax category avoids confusion between different perceptions of values and will be relevant when looking for equivalences later.

$$\boxed{\underset{e \Downarrow_S u}{\mathsf{fwd}}\ (\mathring{e}) = \mathring{u}}$$

$$\text{BS-A}\ \frac{\mathring{e} \sqsubseteq n}{\underset{n \Downarrow_S n}{\mathsf{fwd}}\ (\mathring{e}) = \mathring{e}} \qquad \text{BS-B}\ \frac{\underset{e_1 \Downarrow_S n_1}{\mathsf{fwd}}\ (\mathring{e}_1) = n_1 \qquad \underset{e_2 \Downarrow_S n_2}{\mathsf{fwd}}\ (\mathring{e}_2) = n_2}{\underset{e_1 + e_2 \Downarrow_S n_1 \hat{+} n_2}{\mathsf{fwd}}\ (\mathring{e}_1 + \mathring{e}_2) = n_1 \hat{+} n_2}$$

$$\text{BS-C}\ \frac{\underset{e_1 \Downarrow_S n_1}{\mathsf{fwd}}\ (\mathring{e}_1) = \mathring{e}_1' \qquad \underset{e_2 \Downarrow_S n_2}{\mathsf{fwd}}\ (\mathring{e}_2) = \mathring{e}_2' \qquad \mathring{e}_1' = \blacksquare \text{ or } \mathring{e}_2' = \blacksquare}{\underset{e_1 + e_2 \Downarrow_S n_1 \hat{+} n_2}{\mathsf{fwd}}\ (\mathring{e}_1 + \mathring{e}_2) = \blacksquare}$$

$$\text{BS-D}\ \frac{\underset{e_1 \Downarrow_S n_1}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare \qquad \underset{e_2 \Downarrow_S n_2}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare}{\underset{e_1 + e_2 \Downarrow_S n_1 \hat{+} n_2}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare}$$

$$\text{BS-E}\ \frac{}{\underset{\mathsf{rec}\ f(x).e \Downarrow_S \mathsf{rec}\ f(x).e}{\mathsf{fwd}}\ (\mathsf{rec}\ f(x)\,.\,\mathring{e}) = \mathsf{rec}\ f(x)\,.\,\mathring{e}} \qquad \text{BS-F}\ \frac{}{\underset{\mathsf{rec}\ f(x).e \Downarrow_S \mathsf{rec}\ f(x).e}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare}$$

$$\text{BS-G}\ \frac{\begin{array}{c}\underset{e_1 \Downarrow_S u_1}{\mathsf{fwd}}\ (\mathring{e}_1) = \mathring{u}_1 \qquad \underset{e_2 \Downarrow_S u_2}{\mathsf{fwd}}\ (\mathring{e}_2) = \mathring{u}_2 \qquad u_1 = \mathsf{rec}\ f(x)\,.\,e \\ \underset{[u_1/f][u_2/x]e \Downarrow_S u}{\mathsf{fwd}}\ ([\mathring{u}_1/f][\mathring{u}_2/x]\mathring{e}) = \mathring{u} \qquad \mathring{u}_1 = \mathsf{rec}\ f(x)\,.\,\mathring{e}\end{array}}{\underset{e_1\ e_2 \Downarrow_S u}{\mathsf{fwd}}\ (\mathring{e}_1\ \mathring{e}_2) = \mathring{u}}$$

$$\text{BS-H}\ \frac{\begin{array}{c}\underset{e_1 \Downarrow_S u_1}{\mathsf{fwd}}\ (\mathring{e}_1) = \blacksquare \qquad \underset{e_2 \Downarrow_S u_2}{\mathsf{fwd}}\ (\mathring{e}_2) = \mathring{u}_2 \qquad u_1 = \mathsf{rec}\ f(x)\,.\,e \\ \underset{[u_1/f][u_2/x]e \Downarrow_S u}{\mathsf{fwd}}\ ([\blacksquare/f][\blacksquare/x]\blacksquare) = \blacksquare\end{array}}{\underset{e_1\ e_2 \Downarrow_S u}{\mathsf{fwd}}\ (\mathring{e}_1\ \mathring{e}_2) = \blacksquare}$$

$$\text{BS-I}\ \frac{\begin{array}{c}\underset{e_1 \Downarrow_S u_1}{\mathsf{fwd}}\ (\mathring{e}_1) = \blacksquare \qquad \underset{e_2 \Downarrow_S u_2}{\mathsf{fwd}}\ (\mathring{e}_2) = \blacksquare \qquad u_1 = \mathsf{rec}\ f(x)\,.\,e \\ \underset{[u_1/f][u_2/x]e \Downarrow_S u}{\mathsf{fwd}}\ ([\blacksquare/f][\blacksquare/x]\blacksquare) = \blacksquare\end{array}}{\underset{e_1\ e_2 \Downarrow_S u}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare}$$

Figure C.2: Big-step substitution-based semantics

Theorem 46 (Big-step equivalence from environments to substitution)**.** *For any environment-based big-step forward slicing derivation, we can apply the input substitution to the input term (and slice) and there is a substitution-based big-step forward slicing derivation for it, and the outputs of both forward slicing derivations are "expression-equivalent" (in this context, expression equivalent means that the value from the first derivation converts to the expression-value on the second derivation).*

Formally, Let $\underset{\rho, e \Downarrow_\rho v}{\mathsf{fwd}}\ (\mathring{\rho}, \mathring{e}) = \mathring{v}$ *and* $\mathsf{closed}(v)$*. Then for* $\underset{e[\rho]=e'}{\mathsf{subst}}\ (\mathring{e}, \mathring{\rho}) = \mathring{e}'$*, there exists* $\underset{e' \Downarrow_S u}{\mathsf{fwd}}\ (\mathring{e}') = \mathring{u}$ *and* $\lfloor \mathring{v}, v \rfloor = (\mathring{u}, u)$*.*

Proof. By induction on the **depth of** derivation of forward slicing with environments, and by cases on the applicable rules from Figure C.1.

The key step is the case for rule [BE-I], where we must build a derivation using [BS-G]. We can apply the induction hypothesis on the first two premises of the [BE-I]-based derivation to obtain expression-equivalent values. By Lemma 19, we can change the mappings for f and x in the third forward slicing premise with these new values. By Lemma 20, we can split the environment after these two mappings and transform the derivation to obtain an environment-based derivation that applies substitution for f and x instead. We can apply the induction hypothesis on this new derivation to obtain the missing third premise for [BS-G]. This is valid because, by Lemma 21, both derivations have the same depth. □

Lemma 22 (Single-step slicing can be plugged on a frame)**.** *For any reduction* $\underset{e \longmapsto e'}{\mathsf{fwd}}\ (\mathring{e}) = \mathring{e}'$ *and a pair of a frame and frame slice* $\mathring{F} \sqsubseteq F$*, we can build a derivation* $\underset{F[e] \longmapsto F[e']}{\mathsf{fwd}}\ \left(\mathring{F}[\mathring{e}]\right) = \mathring{F}[\mathring{e}']$

Proof. By cases on each possible frame, we can use rules [SS-G], [SS-I], [SS-K], or [SS-M], respectively. □

Lemma 23 (Multi-step slicing can be plugged on a frame)**.** *For any reduction* $\underset{e \longmapsto^* e'}{\mathsf{fwd}}\ (\mathring{e}) = \mathring{e}'$ *and a pair of a frame and frame slice* $\mathring{F} \sqsubseteq F$*, we can build a derivation* $\underset{F[e] \longmapsto^* F[e']}{\mathsf{fwd}}\ \left(\mathring{F}[\mathring{e}]\right) = \mathring{F}[\mathring{e}']$

Proof. By induction on the derivation of multi-step reduction. For base case [MS-A], we can build a derivation using [MS-A]. For the inductive case [MS-B], we can build a new derivation using [MS-B] using Lemma 22 in the first premise and the induction hypothesis on the second premise.

□

Lemma 24 (Small-step transitivity)**.** *For any* $\underset{e \longmapsto^* e'}{\mathsf{fwd}}\ (\mathring{e}) = \mathring{e}'$ *and* $\underset{e' \longmapsto^* e''}{\mathsf{fwd}}\ (\mathring{e}') = \mathring{e}''$*, it is also the case that* $\underset{e \longmapsto^* e''}{\mathsf{fwd}}\ (\mathring{e}) = \mathring{e}''$*.*

Proof. By structural induction on the first small-step derivation. □

Theorem 47 (Big-step to small-step equivalence)**.** *For any* $\underset{e \Downarrow_S u}{\mathsf{fwd}}\ (\mathring{e}) = \mathring{u}$*, there exists also a derivation* $\underset{e \longmapsto^* u}{\mathsf{fwd}}\ (\mathring{e}) = \mathring{u}$*.*

$$\boxed{\underset{e \longmapsto e}{\mathsf{fwd}}\ (\mathring{e}) = \mathring{e}}$$

$$\text{SS-A}\ \frac{}{\underset{(\mathsf{rec}\ f(x).e)\ u \longmapsto [\mathsf{rec}\ f(x).e/f][u/x]e}{\mathsf{fwd}}\ ((\mathsf{rec}\ f(x)\ .\ \mathring{e})\ \mathring{u}) = \mathsf{rec}\ f(x)\ .\ \mathring{e}/f][\mathring{u}/x]\mathring{e}}$$

$$\text{SS-B}\ \frac{}{\underset{(\mathsf{rec}\ f(x).e)\ u \longmapsto [\mathsf{rec}\ f(x).e/f][u/x]e}{\mathsf{fwd}}\ (\blacksquare\ \mathring{u}) = \blacksquare/f][\blacksquare/x]\blacksquare}$$

$$\text{SS-C}\ \frac{}{\underset{(\mathsf{rec}\ f(x).e)\ u \longmapsto [\mathsf{rec}\ f(x).e/f][u/x]e}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare/f][\blacksquare/x]\blacksquare}$$

$$\text{SS-D}\ \frac{}{\underset{n_1+n_2 \longmapsto n_1 \hat{+} n_2}{\mathsf{fwd}}\ (n_1 + n_2) = n_1 \hat{+} n_2} \qquad \text{SS-E}\ \frac{\mathring{e}_1 = \blacksquare \text{ or } \mathring{e}_2 = \blacksquare}{\underset{n_1+n_2 \longmapsto n_1 \hat{+} n_2}{\mathsf{fwd}}\ (\mathring{e}_1 + \mathring{e}_2) = \blacksquare}$$

$$\text{SS-F}\ \frac{}{\underset{n_1+n_2 \longmapsto n_1 \hat{+} n_2}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare} \qquad \text{SS-G}\ \frac{\underset{e_1 \longmapsto e_1'}{\mathsf{fwd}}\ (\mathring{e}_1) = \mathring{e}_1'}{\underset{e_1\ e_2 \longmapsto e_1'\ e_2}{\mathsf{fwd}}\ (\mathring{e}_1\ \mathring{e}_2) = \mathring{e}_1'\ \mathring{e}_2}$$

$$\text{SS-H}\ \frac{\underset{e_1 \longmapsto e_1'}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare}{\underset{e_1\ e_2 \longmapsto e_1'\ e_2}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare} \qquad \text{SS-I}\ \frac{\underset{e_2 \longmapsto e_2'}{\mathsf{fwd}}\ (\mathring{e}_2) = \mathring{e}_2'}{\underset{u_1\ e_2 \longmapsto u_1\ e_2'}{\mathsf{fwd}}\ (\mathring{u}_1\ \mathring{e}_2) = \mathring{u}_1\ \mathring{e}_2'}$$

$$\text{SS-J}\ \frac{\underset{e_2 \longmapsto e_2'}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare}{\underset{u_1\ e_2 \longmapsto u_1\ e_2'}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare} \qquad \text{SS-K}\ \frac{\underset{e_1 \longmapsto e_1'}{\mathsf{fwd}}\ (\mathring{e}_1) = \mathring{e}_1'}{\underset{e_1+e_2 \longmapsto e_1'+e_2}{\mathsf{fwd}}\ (\mathring{e}_1 + \mathring{e}_2) = \mathring{e}_1' + \mathring{e}_2}$$

$$\text{SS-L}\ \frac{\underset{e_1 \longmapsto e_1'}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare}{\underset{e_1+e_2 \longmapsto e_1'+e_2}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare} \qquad \text{SS-M}\ \frac{\underset{e_2 \longmapsto e_2'}{\mathsf{fwd}}\ (\mathring{e}_2) = \mathring{e}_2'}{\underset{u_1+e_2 \longmapsto u_1+e_2'}{\mathsf{fwd}}\ (\mathring{u}_1 + \mathring{e}_2) = \mathring{u}_1 + \mathring{e}_2'}$$

$$\text{SS-N}\ \frac{\underset{e_2 \longmapsto e_2'}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare}{\underset{u_1+e_2 \longmapsto u_1+e_2'}{\mathsf{fwd}}\ (\blacksquare) = \blacksquare}$$

$$\boxed{\underset{e \longmapsto^* e}{\mathsf{fwd}}\ (\mathring{e}) = \mathring{e}}$$

$$\text{MS-A}\ \frac{}{\underset{e \longmapsto^* e}{\mathsf{fwd}}\ (\mathring{e}) = \mathring{e}} \qquad \text{MS-B}\ \frac{\underset{e \longmapsto e'}{\mathsf{fwd}}\ (\mathring{e}) = \mathring{e}' \qquad \underset{e' \longmapsto^* e''}{\mathsf{fwd}}\ (\mathring{e}') = \mathring{e}''}{\underset{e \longmapsto^* e''}{\mathsf{fwd}}\ (\mathring{e}) = \mathring{e}''}$$

Figure C.3: Small-step substitution-based semantics

Proof. By structural induction on the derivation of big-step reduction with substitution.

The interesting case is [BS-G], all other cases are similar or simpler. By induction hypothesis, we can obtain a small-step reduction for both premises. On each we can apply Lemma 23 to obtain a new derivation with the right frames ($\square\ \mathring{e}_2 \sqsubseteq \square\ e_2$ for the first and $\mathring{u}_1\ \square \sqsubseteq u_1\ \square$ for the second). These two can be combined by Lemma 24 to obtain a derivation $\underset{e_1\ e_2 \Downarrow_S (\mathsf{rec}\ f(x).e)\ u_2}{\mathsf{fwd}} (\mathring{e}_1\ \mathring{e}_2) = (\mathsf{rec}\ f(x)\ .\ \mathring{e})\ \mathring{u}_2$. Using rule [MS-B] with premises [SS-A] and [MS-A], we can build another derivation that steps the beta-reduction step, which can be combined with the previous derivation again by Lemma 24. Finally, we can use again Lemma 24 to combine that derivation with the derivation obtained from applying the induction hypothesis on the last inductive premise of our [BS-G] hypothesis (for the post-substitution term). $\square$

C.2 ADDING STORES

I now introduce the extra rules for slicing with stores. All the rules in the previous section are reused, and they pass the store to their premises and propagate their outputs whenever the domains of functions change.

First, we extend our language with locations l, $\mathsf{alloc}\ e$, $\mathsf{deref}\ e$ $\mathsf{set!}\ e\ e$. Then we extend the slices. Our slicing functions are extended HERE

We then extend the auxiliary operations:

Definition 24 (Values to Expressions conversion, extended)**.** *We extend conversion from values to expressions to deal with memory locations:*

$$\frac{\mathring{e} \sqsubseteq l}{\lfloor \mathring{e}, l \rfloor = (\mathring{e}, l)}$$

And we can move on to prove the extended version of lemmas:

Lemma 25 (Closed values are substitution-immune, extended)**.** *Let $\lfloor \mathring{v}, v \rfloor = (\mathring{e}, e)$ and $\mathsf{closed}(v)$. For any environments,*

$$\underset{e[\rho]=e'}{\mathsf{subst}}\ (\mathring{e}, \mathring{\rho}) = \mathring{e}' \textit{ iff } \underset{e[\rho']=e'}{\mathsf{subst}}\ \left(\mathring{e}, \mathring{\rho}'\right) = \mathring{e}'$$

Proof. Same structure of proof as before. Base case for l is trivial. Rest of cases for new expressions follow by induction hypothesis. $\square$

Lemma 26 (Expression-equivalent environment changes produce expression-equivalent values)**.** *Let $\underset{\rho, e, \mu \Downarrow_\rho v, \mu'}{\mathsf{fwd}} (\mathring{\rho}, \mathring{e}, \mathring{\mu}) = \mathring{v}, \mathring{\mu}'$, for any $\mathring{\rho}' \sqsubseteq \rho'$ such that the domains of ρ and ρ' are equivalent and their mappings (with their slices) are expression-equivalent, then there exists an equivalent trace that produces an output that is expression-equivalent to the output of the original trace, and produces an expression-equivalent store. That is, there exists $\underset{\rho', e, \mu \Downarrow_\rho v', \mu''}{\mathsf{fwd}} (\mathring{\rho}', \mathring{e}, \mathring{\mu}) = \mathring{v}', \mathring{\mu}''$ and*

$$\lfloor \mathring{v}, v \rfloor = (\mathring{e}_v, e_v) \textit{ iff } \lfloor \mathring{v}', v' \rfloor = (\mathring{e}_v, e_v)$$

$$\boxed{\underset{\rho,e,\mu\Downarrow_\rho v,\mu}{\mathsf{fwd}}\ (\mathring{\rho},\mathring{e},\mathring{\mu}) = \mathring{v},\mathring{\mu}}$$

$$\text{BE-L}\ \frac{\underset{\rho,e,\mu\Downarrow_\rho v,\mu'}{\mathsf{fwd}}\ (\mathring{\rho},\mathring{e},\mathring{\mu}) = \mathring{v},\mathring{\mu}' \qquad l \notin dom(\mu')}{\underset{\rho,\mathsf{alloc}\ v,\mu\Downarrow_\rho \rho,l,[l\mapsto v]\mu'}{\mathsf{fwd}}\ (\mathring{\rho},\mathsf{alloc}\ \mathring{e},\mathring{\mu}) = \mathring{\rho},l,[l\mapsto \mathring{v}]\mathring{\mu}'}$$

$$\text{BE-M}\ \frac{\underset{\rho,e,\mu\Downarrow_\rho v,\mu'}{\mathsf{fwd}}\ (\mathring{\rho},\blacksquare,\mathring{\mu}) = \blacksquare,\mathring{\mu}' \qquad l \notin dom(\mu')}{\underset{\rho,\mathsf{alloc}\ v,\mu\Downarrow_\rho \rho,l,[l\mapsto v]\mu'}{\mathsf{fwd}}\ (\mathring{\rho},\blacksquare,\mathring{\mu}) = \mathring{\rho},\blacksquare,[l\mapsto \blacksquare]\mathring{\mu}'}$$

$$\text{BE-N}\ \frac{\underset{\rho,e,\mu\Downarrow_\rho l,\mu'}{\mathsf{fwd}}\ (\mathring{\rho},\mathring{e},\mathring{\mu}) = l,\mathring{\mu}' \qquad l \in dom(\mu')}{\underset{\rho,\mathsf{deref}\ e,\mu\Downarrow_\rho \mu'(l),\mu'}{\mathsf{fwd}}\ (\mathring{\rho},\mathsf{deref}\ \mathring{e},\mathring{\mu}) = \mathring{\mu}'(l),\mathring{\mu}'}$$

$$\text{BE-O}\ \frac{\underset{\rho,e,\mu\Downarrow_\rho l,\mu'}{\mathsf{fwd}}\ (\mathring{\rho},\mathring{e},\mathring{\mu}) = \blacksquare,\mathring{\mu}' \qquad l \in dom(\mu')}{\underset{\rho,\mathsf{deref}\ e,\mu\Downarrow_\rho \mu'(l),\mu'}{\mathsf{fwd}}\ (\mathring{\rho},\mathsf{deref}\ \mathring{e},\mathring{\mu}) = \blacksquare,\mathring{\mu}'}$$

$$\text{BE-P}\ \frac{\underset{\rho,e,\mu\Downarrow_\rho l,\mu'}{\mathsf{fwd}}\ (\mathring{\rho},\blacksquare,\mathring{\mu}) = \blacksquare,\mathring{\mu}' \qquad l \in dom(\mu')}{\underset{\rho,\mathsf{deref}\ e,\mu\Downarrow_\rho \mu'(l),\mu'}{\mathsf{fwd}}\ (\mathring{\rho},\blacksquare,\mathring{\mu}) = \blacksquare,\mathring{\mu}'}$$

$$\text{BE-Q}\ \frac{\underset{\rho,e_1,\mu\Downarrow_\rho l,\mu'}{\mathsf{fwd}}\ (\mathring{\rho},\mathring{e}_1,\mathring{\mu}) = l,\mathring{\mu}' \qquad \underset{\rho,e_2,\mu'\Downarrow_\rho v,\mu''}{\mathsf{fwd}}\ (\mathring{\rho},\mathring{e}_2,\mathring{\mu}') = \mathring{v},\mathring{\mu}'' \qquad l \in dom(\mu')}{\underset{\rho,\mathsf{set!}\ e_1\ e_2,\mu\Downarrow_\rho v,[l\mapsto v]\mu''}{\mathsf{fwd}}\ (\mathring{\rho},\mathsf{set!}\ \mathring{e}_1\ \mathring{e}_2,\mathring{\mu}) = \mathring{v},[l\mapsto \mathring{v}]\mathring{\mu}''}$$

$$\text{BE-R}\ \frac{\underset{\rho,e_1,\mu\Downarrow_\rho l,\mu'}{\mathsf{fwd}}\ (\mathring{\rho},\mathring{e}_1,\mathring{\mu}) = \blacksquare,\mathring{\mu}' \qquad \underset{\rho,e_2,\mu'\Downarrow_\rho v,\mu''}{\mathsf{fwd}}\ (\mathring{\rho},\mathring{e}_2,\mathring{\mu}') = \mathring{v},\mathring{\mu}'' \qquad l \in dom(\mu')}{\underset{\rho,\mathsf{set!}\ e_1\ e_2,\mu\Downarrow_\rho v,[l\mapsto v]\mu''}{\mathsf{fwd}}\ (\mathring{\rho},\mathsf{set!}\ \mathring{e}_1\ \mathring{e}_2,\mathring{\mu}) = \blacksquare,[l\mapsto \blacksquare]\mathring{\mu}''}$$

$$\text{BE-S}\ \frac{\underset{\rho,e_1,\mu\Downarrow_\rho l,\mu'}{\mathsf{fwd}}\ (\mathring{\rho},\blacksquare,\mathring{\mu}) = \blacksquare,\mathring{\mu}' \qquad \underset{\rho,e_2,\mu'\Downarrow_\rho v,\mu''}{\mathsf{fwd}}\ (\mathring{\rho},\blacksquare,\mathring{\mu}') = \blacksquare,\mathring{\mu}'' \qquad l \in dom(\mu')}{\underset{\rho,\mathsf{set!}\ e_1\ e_2,\mu\Downarrow_\rho v,[l\mapsto v]\mu''}{\mathsf{fwd}}\ (\mathring{\rho},\blacksquare,\mathring{\mu}) = \blacksquare,[l\mapsto \blacksquare]\mathring{\mu}''}$$

Figure C.4: Big-step environment based semantics, store extensions

$$\textit{and for every } l \in dom(\mu), \quad \lfloor \mathring{\mu}'(l), \mu'(l) \rfloor = (\mathring{e}_v, e_v) \textit{ iff } \lfloor \mathring{\mu}''(l), \mu''(l) \rfloor = (\mathring{e}_v, e_v)$$

Proof. The proof for the non-state cases don't change, as stores are only passed around. Thus by induction hypothesis the expression-equivalent store part in those cases holds.

In the cases that mutate the store, the updated to the store are expression-equivalent by induction hypothesis, and the outgoing values remain expression-equivalent. □

Lemma 27 (Environment substitution preserves an expression-equivalent output value, extended)**.** *Let* $\underset{\rho_1 +\!\!+ \rho_2, e, \mu \Downarrow_\rho v, \mu'}{\mathsf{fwd}} (\mathring{\rho}_1 +\!\!+ \mathring{\rho}_2, \mathring{e}, \mathring{\mu}) = \mathring{v}, \mathring{\mu}'$ *and* $\mathsf{closed}(v)$.

When the substitution is partially applied, there is a big-step evaluation trace that preserves expression-equivalence of the outcoming values. That is,

$$\textit{for } \underset{e[\rho_1]=e'}{\mathsf{subst}} (\mathring{e}, \mathring{\rho}_1) = \mathring{e}' \textit{ and } \underset{\rho_2, e', \mu \Downarrow_\rho v', \mu''}{\mathsf{fwd}} (\mathring{\rho}_2, \mathring{e}', \mathring{\mu}) = \mathring{v}', \mathring{\mu}'' \textit{ then } \textbf{both}$$

$$\lfloor \mathring{v}, v \rfloor = (\mathring{e}_v, e_v) \textit{ iff } \lfloor \mathring{v}', v' \rfloor = (\mathring{e}_v, e_v)$$

$$\textit{and for every } l \in dom(\mu), \quad \lfloor \mathring{\mu}'(l), \mu'(l) \rfloor = (\mathring{e}_v, e_v) \textit{ iff } \lfloor \mathring{\mu}''(l), \mu''(l) \rfloor = (\mathring{e}_v, e_v)$$

Proof. Substitution does not interact with store operations, so the proof follows by the same structure as without stores. For the cases dealing with the store, the induction preserves updates to the store to ensure that expression-equivalence holds for both the output values and the store. □

Definition 25 (Extended expression-values)**.** *We extend the definition of expression-values to deal with locations:*

Syntax

$$\begin{array}{rcll} u \in \textsc{ExprValue} & ::= & n \mid \mathsf{rec}\ f(x)\,.\,e \mid l & \textit{(expression-values)} \\ \mathring{u} \in \textsc{ExprValueSlice} & ::= & n \mid \mathsf{rec}\ f(x)\,.\,\mathring{e} \mid l \mid \blacksquare & \textit{(expression-value slices)} \end{array}$$

Note that the stores used in the substitution-based semantices map locations to expression-values.

Theorem 48 (Big-step equivalence from environments to substitution)**.** *For any environment-based big-step forward slicing derivation, we can apply the input substitution to the input term (and slice) and there is a substitution-based big-step forward slicing derivation for it, and the outputs of both forward slicing derivations are "expression-equivalent" (in this context, expression equivalent means that the value from the first derivation converts to the expression-value on the second derivation).*

Formally, Let $\underset{\rho, e, \mu \Downarrow_\rho v, \mu'}{\mathsf{fwd}} (\mathring{\rho}, \mathring{e}, \mathring{\mu}) = \mathring{v}, \mathring{\mu}'$ *and* $\mathsf{closed}(v)$. *Then for* $\underset{e[\rho]=e'}{\mathsf{subst}} (\mathring{e}, \mathring{\rho}) = \mathring{e}'$, *there exists* $\underset{e', \mu \Downarrow_S u, \mu''}{\mathsf{fwd}} (\mathring{e}', \mathring{\mu}) = \mathring{u}, \mathring{\mu}''$ *such that*

$$\lfloor \mathring{v}, v \rfloor = (\mathring{u}, u)$$

$$dom(\mu') = dom(\mu'')$$

$$\textit{and for every } l \in dom(\mu'), \quad \lfloor \mathring{\mu}'(l), \mu'(l) \rfloor = \big(\mathring{\mu}''(l), \mu''(l)\big)$$

$$\boxed{\underset{e,\mu \Downarrow_S u,\mu}{\mathsf{fwd}}\ (\mathring{e}, \mathring{\mu}) = \mathring{u}, \mathring{\mu}}$$

$$\text{BS-J}\ \frac{\underset{e,\mu \Downarrow_S v,\mu'}{\mathsf{fwd}}\ (\mathring{e}, \mathring{\mu}) = \mathring{v}, \mathring{\mu}' \qquad l \notin dom(\mu')}{\underset{\mathsf{alloc}\ v,\mu \Downarrow_S l,[l \mapsto v]\mu'}{\mathsf{fwd}}\ (\mathsf{alloc}\ \mathring{e}, \mathring{\mu}) = l, [l \mapsto \mathring{v}]\mathring{\mu}'}$$

$$\text{BS-K}\ \frac{\underset{e,\mu \Downarrow_S v,\mu'}{\mathsf{fwd}}\ (\blacksquare, \mathring{\mu}) = \blacksquare, \mathring{\mu}' \qquad l \notin dom(\mu')}{\underset{\mathsf{alloc}\ v,\mu \Downarrow_S l,[l \mapsto v]\mu'}{\mathsf{fwd}}\ (\blacksquare, \mathring{\mu}) = \blacksquare, [l \mapsto \blacksquare]\mathring{\mu}'}$$

$$\text{BS-L}\ \frac{\underset{e,\mu \Downarrow_S l,\mu'}{\mathsf{fwd}}\ (\mathring{e}, \mathring{\mu}) = l, \mathring{\mu}' \qquad l \in dom(\mu')}{\underset{\mathsf{deref}\ e,\mu \Downarrow_S \mu'(l),\mu'}{\mathsf{fwd}}\ (\mathsf{deref}\ \mathring{e}, \mathring{\mu}) = \mathring{\mu}'(l), \mathring{\mu}'}$$

$$\text{BS-M}\ \frac{\underset{e,\mu \Downarrow_S l,\mu'}{\mathsf{fwd}}\ (\mathring{e}, \mathring{\mu}) = \blacksquare, \mathring{\mu}' \qquad l \in dom(\mu')}{\underset{\mathsf{deref}\ e,\mu \Downarrow_S \mu'(l),\mu'}{\mathsf{fwd}}\ (\mathsf{deref}\ \mathring{e}, \mathring{\mu}) = \blacksquare, \mathring{\mu}'}$$

$$\text{BS-N}\ \frac{\underset{e,\mu \Downarrow_S l,\mu'}{\mathsf{fwd}}\ (\blacksquare, \mathring{\mu}) = \blacksquare, \mathring{\mu}' \qquad l \in dom(\mu')}{\underset{\mathsf{deref}\ e,\mu \Downarrow_S \mu'(l),\mu'}{\mathsf{fwd}}\ (\blacksquare, \mathring{\mu}) = \blacksquare, \mathring{\mu}'}$$

$$\text{BS-O}\ \frac{\underset{e_1,\mu \Downarrow_S l,\mu'}{\mathsf{fwd}}\ (\mathring{e}_1, \mathring{\mu}) = l, \mathring{\mu}' \qquad \underset{e_2,\mu' \Downarrow_S v,\mu''}{\mathsf{fwd}}\ (\mathring{e}_2, \mathring{\mu}') = \mathring{v}, \mathring{\mu}'' \qquad l \in dom(\mu')}{\underset{\mathsf{set!}\ e_1\ e_2,\mu \Downarrow_S v,[l \mapsto v]\mu''}{\mathsf{fwd}}\ (\mathsf{set!}\ \mathring{e}_1\ \mathring{e}_2, \mathring{\mu}) = \mathring{v}, [l \mapsto \mathring{v}]\mathring{\mu}''}$$

$$\text{BS-P}\ \frac{\underset{e_1,\mu \Downarrow_S l,\mu'}{\mathsf{fwd}}\ (\mathring{e}_1, \mathring{\mu}) = \blacksquare, \mathring{\mu}' \qquad \underset{e_2,\mu' \Downarrow_S v,\mu''}{\mathsf{fwd}}\ (\mathring{e}_2, \mathring{\mu}') = \mathring{v}, \mathring{\mu}'' \qquad l \in dom(\mu')}{\underset{\mathsf{set!}\ e_1\ e_2,\mu \Downarrow_S v,[l \mapsto v]\mu''}{\mathsf{fwd}}\ (\mathsf{set!}\ \mathring{e}_1\ \mathring{e}_2, \mathring{\mu}) = \blacksquare, [l \mapsto \blacksquare]\mathring{\mu}''}$$

$$\text{BS-Q}\ \frac{\underset{e_1,\mu \Downarrow_S l,\mu'}{\mathsf{fwd}}\ (\blacksquare, \mathring{\mu}) = \blacksquare, \mathring{\mu}' \qquad \underset{e_2,\mu' \Downarrow_S v,\mu''}{\mathsf{fwd}}\ (\blacksquare, \mathring{\mu}') = \blacksquare, \mathring{\mu}'' \qquad l \in dom(\mu')}{\underset{\mathsf{set!}\ e_1\ e_2,\mu \Downarrow_S v,[l \mapsto v]\mu''}{\mathsf{fwd}}\ (\mathsf{set!}\ \mathring{e}_1\ \mathring{e}_2, \mathring{\mu}) = \blacksquare, [l \mapsto \blacksquare]\mathring{\mu}''}$$

Figure C.5: Big-step substitution-based semantics, store extensions

Proof. Most of the proof is the same as without stores. Operations that deal with store do not apply substitution, so everything follows by applying induction hypotheses on the smaller-depth premises. □

For the definition of $\longmapsto$ for state-based rules, we refer to the rules in Figure 4.10. The frame rules follow the inductive structure. Multi-step reduction remains the same *plus store propagation*, so Small-step transitivity still holds.

Lemma 28 (Single-step slicing can be plugged on a frame,extended)**.** *For any reduction* $\underset{e,\mu \longmapsto e',\mu'}{\mathsf{fwd}} (\mathring{e}, \mathring{\mu}) = \mathring{e}', \mathring{\mu}'$ *and a pair of a frame and frame slice* $\mathring{F} \sqsubseteq F$*, we can build a derivation* $\underset{F[e],\mu \longmapsto F[e'],\mu'}{\mathsf{fwd}} \left(\mathring{F}[\mathring{e}], \mathring{\mu}\right) = \mathring{F}[\mathring{e}'], \mathring{\mu}'$

Lemma 29 (Multi-step slicing can be plugged on a frame)**.**
For any reduction $\underset{e,\mu \longmapsto^* e',\mu'}{\mathsf{fwd}} (\mathring{e}, \mathring{\mu}) = \mathring{e}', \mathring{\mu}'$ *and a pair of a frame and frame slice* $\mathring{F} \sqsubseteq F$*, we can build a derivation* $\underset{F[e],\mu \longmapsto^* F[e'],\mu'}{\mathsf{fwd}} \left(\mathring{F}[\mathring{e}], \mathring{\mu}\right) = \mathring{F}[\mathring{e}'], \mathring{\mu}'$

Proof. Just like before, but using Lemma 28 instead of Lemma 22. □

Theorem 49 (Big-step to small-step equivalence, extended)**.** *For any* $\underset{e,\mu \Downarrow_S u,\mu'}{\mathsf{fwd}} (\mathring{e}, \mathring{\mu}) = \mathring{u}, \mathring{\mu}'$*, there exists also a derivation* $\underset{e,\mu \longmapsto^* u,\mu'}{\mathsf{fwd}} (\mathring{e}, \mathring{\mu}) = \mathring{u}, \mathring{\mu}'$*.*

Proof. The proof is just like for Theorem 47. The rules that update the store do so in equivalent ways in both, and the same techniques as before (thanks to the extended Lemma 29) can be applied. □

D DEFINING EVIDENCE COMPOSITION INDUCTIVELY

To account for only well-formed evidence, I can treat evidence objects with the following syntax:

$$
\begin{array}{rcll}
\varepsilon \in \mathrm{Ev}^{<:} & ::= & \langle \mathsf{Int},\mathsf{Int} \rangle \mid \langle \mathsf{Bool},\mathsf{Bool} \rangle \mid \langle ?, ? \rangle \mid & \text{(evidence objects)} \\
 & & \varepsilon \to \varepsilon \mid \left[\sum_{i=1}^{n} \ell_i : \mathcal{M}_i \left(*_1, *_2 \right) \right] & \\
\mathcal{M} \in \textsc{EvMapping} & ::= & \langle M, \varnothing \rangle \mid \varepsilon_{(R,R)} \mid \langle S, \varepsilon \rangle_{(*,O)} & \text{(evidence mappings)} \\
\varepsilon_\perp \in \mathrm{Ev}_\perp^{<:} & ::= & \varepsilon \mid \perp & \text{(possibly failing evidence objects)} \\
\mathcal{M}_\perp \in \textsc{EvMapping}_\perp & ::= & \mathcal{M} \mid \perp & \text{(possibly failing evidence mappings)}
\end{array}
$$

$\boxed{\varepsilon ; \varepsilon = \varepsilon_\perp}$

$$
\frac{}{[(?,?)] ; [(*_3,*_4)] = [(?,*_4)]}
\qquad
\frac{}{[(*_1,\cdot)] ; [(*_3,*_4)] = [(*_1,\cdot)]}
$$

$$
\frac{\begin{array}{c}\mathcal{M}_{11} ; \mathcal{M}_{12} = \mathcal{M}_{13} \\ \left[\sum_{i=2}^{n} \ell_i : \mathcal{M}_{i1} \left(*_1, *_2 \right) \right] ; \left[\sum_{i=2}^{n} \ell_i : \mathcal{M}_{i2} \left(*_3, *_4 \right) \right] = \left[\sum_{i=2}^{n} \ell_i : \mathcal{M}_{i3} \left(*_5, *_6 \right) \right]\end{array}}{\left[\sum_{i=1}^{n} \ell_i : \mathcal{M}_{i1} \left(*_1, *_2 \right) \right] ; \left[\sum_{i=1}^{n} \ell_i : \mathcal{M}_{i2} \left(*_3, *_4 \right) \right] = \left[\sum_{i=1}^{n} \ell_i : \mathcal{M}_{i3} \left(*_5, *_6 \right) \right]}
$$

$$
\frac{\mathcal{M}_{11} ; \mathcal{M}_{12} = \mathcal{M}_{13} \qquad \left[\sum_{i=2}^{n} \ell_i : \mathcal{M}_{i1} \left(*_1, *_2 \right) \right] ; \left[\sum_{i=2}^{n} \ell_i : \mathcal{M}_{i2} \left(*_3, *_4 \right) \right] = \perp}{\left[\sum_{i=1}^{n} \ell_i : \mathcal{M}_{i1} \left(*_1, *_2 \right) \right] ; \left[\sum_{i=1}^{n} \ell_i : \mathcal{M}_{i2} \left(*_3, *_4 \right) \right] = \perp}
$$

$$
\frac{\begin{array}{c}\mathcal{M}_{11} ; \mathcal{M}_{12} = \perp \\ \left[\sum_{i=2}^{n} \ell_i : \mathcal{M}_{i1} \left(*_1, *_2 \right) \right] ; \left[\sum_{i=2}^{n} \ell_i : \mathcal{M}_{i2} \left(*_3, *_4 \right) \right] = \left[\sum_{i=2}^{n} \ell_i : \mathcal{M}_{i3} \left(*_5, *_6 \right) \right]\end{array}}{\left[\sum_{i=1}^{n} \ell_i : \mathcal{M}_{i1} \left(*_1, *_2 \right) \right] ; \left[\sum_{i=1}^{n} \ell_i : \mathcal{M}_{i2} \left(*_3, *_4 \right) \right] = \perp}
$$

$$
\frac{\mathcal{M}_{11} ; \mathcal{M}_{12} = \perp \qquad \left[\sum_{i=2}^{n} \ell_i : \mathcal{M}_{i1} \left(*_1, *_2 \right) \right] ; \left[\sum_{i=2}^{n} \ell_i : \mathcal{M}_{i2} \left(*_3, *_4 \right) \right] = \perp}{\left[\sum_{i=1}^{n} \ell_i : \mathcal{M}_{i1} \left(*_1, *_2 \right) \right] ; \left[\sum_{i=1}^{n} \ell_i : \mathcal{M}_{i2} \left(*_3, *_4 \right) \right] = \perp}
$$

$\boxed{\mathcal{M};\mathcal{M} = \mathcal{M}_\perp}$

$$\frac{M_2 \neq S_R}{\langle M_1, \varnothing\rangle; \langle M_2, \varnothing\rangle = \langle M_1, \varnothing\rangle} \qquad \frac{}{\langle M, \varnothing\rangle; \langle S, \varepsilon\rangle_{(O,O)} = \langle M, \varnothing\rangle}$$

$$\frac{}{\langle M_1, \varnothing\rangle; \langle S_R, \varnothing\rangle = \perp} \qquad \frac{}{\langle M_1, \varnothing\rangle; \varepsilon_{(R,R)} = \perp} \qquad \frac{}{\langle M_1, \varnothing\rangle; \langle S, \varepsilon\rangle_{(R,O)} = \perp}$$

$$\frac{\varepsilon_1;\varepsilon_2 = \varepsilon_3}{\varepsilon_{1(R,R)};\varepsilon_{2(R,R)} = \varepsilon_{3(R,R)}} \qquad \frac{\varepsilon_1;\varepsilon_2 = \perp}{\varepsilon_{1(R,R)};\varepsilon_{2(R,R)} = \perp}$$

$$\frac{\pi_2(\varepsilon_1) \sqcap S \text{ is defined}}{\varepsilon_{1(R,R)}; \langle S_*, \varnothing\rangle = \langle \pi_1(\varepsilon_1)_R, \varnothing\rangle} \qquad \frac{\pi_2(\varepsilon_1) \sqcap S \text{ is undefined}}{\varepsilon_{1(R,R)}; \langle S_*, \varnothing\rangle = \perp} \qquad \frac{}{\varepsilon_{1(R,R)}; \langle \varnothing, \varnothing\rangle = \perp}$$

$$\frac{\varepsilon_1;\varepsilon_2 = \varepsilon_3}{\varepsilon_{1(R,R)}; \langle S, \varepsilon_2\rangle_{(*,O)} = \langle \pi_1(\varepsilon_3), \varepsilon_3\rangle_{(R,O)}} \qquad \frac{\varepsilon_1;\varepsilon_2 = \perp}{\varepsilon_{1(R,R)}; \langle S, \varepsilon_2\rangle_{(*,O)} = \perp}$$

$$\frac{}{\langle S_1, \varepsilon_1\rangle_{(*_1,O)}; \langle \varnothing, \varnothing\rangle = \langle S_{1*_1}, \varnothing\rangle} \qquad \frac{\pi_2(\varepsilon_1) \sqcap S_2 \text{ is defined}}{\langle S_1, \varepsilon_1\rangle_{(*_1,O)}; \langle S_{2*_2}, \varnothing\rangle = \langle S_{1*_1}, \varnothing\rangle}$$

$$\frac{\pi_2(\varepsilon_1) \sqcap S_2 \text{ is undefined}}{\langle S_1, \varepsilon_1\rangle_{(*_1,O)}; \langle S_{2*_2}, \varnothing\rangle = \perp} \qquad \frac{\varepsilon_1;\varepsilon_2 = \varepsilon_3}{\langle S_1, \varepsilon_1\rangle_{(*_1,O)}; \varepsilon_{2(R,R)} = \varepsilon_{3(R,R)}}$$

$$\frac{\varepsilon_1;\varepsilon_2 = \perp}{\langle S_1, \varepsilon_1\rangle_{(*_1,O)}; \varepsilon_{2(R,R)} = \perp} \qquad \frac{\varepsilon_1;\varepsilon_2 = \varepsilon_3 \qquad *_1 \sqcap *_2 = *_3}{\langle S_1, \varepsilon_1\rangle_{(*_1,O)}; \varepsilon_{2(*_2,O)} = \langle S_1, \varepsilon_3\rangle_{(*_3,O)}}$$

$$\frac{\varepsilon_1;\varepsilon_2 = \perp}{\langle S_1, \varepsilon_1\rangle_{(*_1,O)}; \varepsilon_{2(*_2,O)} = \langle S_1, \varnothing\rangle}$$

I can use this definitions to provide forward slicing.

E USER STUDY DESIGN

E.1 PRE-SCREENING QUESTIONNAIRE

1. In which programming languages have you written at least 1000 LOC (select all that apply) ?
   - ☐ JavaScript
   - ☐ Python
   - ☐ Swift
   - ☐ TypeScript
   - ☐ Haskell
   - ☐ C#
   - ☐ Others (please specify) _____
2. Which of the following programs are valid TypeScript programs?
   - ☐ Program 1:

     ```
     def example(x : float) -> str :
       print("TypeScript")
       return str(x)
     ```
   - ☐ Program 2:

     ```
     class Example {
       static void Main(string[] args) {
         string msg = "TypeScript";
         Console.WriteLine(msg);
       }
     }
     ```
   - ☐ Program 3:

     ```
     function example (x : number) : { id: string } {
       console.log("TypeScript")
       return { id: x.toString() }
     }
     ```
   - ☐ Program 4:

     ```
     func example(x: Int) -> String {
       print("TypeScript")
       return String(x)
     }
     ```

3. Read the following program and select **all** the statements that correctly apply to it:

```
let person: { id: string } = { id:'hello' }
let parser : (x: string) => any = JSON.parse
let list_1 = parser('[]')

function positive(x): boolean{
  if (x < 0) {
    return false
  }
  return true
}

let y = [1, 'a', 'b', 40]
```

   - ☐ `person` has type `string`
   - ☐ `parser` has type `(x: string) => any`
   - ☐ inside the function `positive`, `x` has type `any`
   - ☐ `list_1` has type `any`
   - ☐ inside the function `positive`, `x` has type `any`
   - ☐ `y` has type `any`
   - ☐ `y` has type `number[]`
   - ☐ `y` has type `(number | string)[]`

E.2 TOOL TUTORIAL

Figures E.1 to E.5 include screenshots from the tutorial presented in the user study.

E.3 MAIN TASK DESCRIPTION

There are minor changes in the interface depending on whether the slicing tool ("TypeSlicer") is available. Figures E.6 to E.8 show the interface and task description when the tool is available, and Figures E.9 and E.10 show the interface and task description when the tool is not available.

E.4 EXIT QUESTIONNAIRE

1. Do you have any comments about the TypeSlicer tool that you would like to share with us? *(Open comment box)*
2. Do you have any comments about the programs or the errors you were faced with that you would like to share with us? *(Open comment box)*

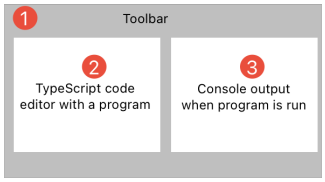
Identifying the causes of runtime type errors with dynamic program slices

Tutorial

Welcome! We are studying TypeSlicer, a tool that enforces type assertions in a TypeScript-like language.

Intro to our interface

For this study, you will be presented with some TypeScript programs in a browser-based code editor. The interface consists of the following elements:



1. **The toolbar** provides options you are expected to interact with during the study.
2. **The code editor** will show the program we are currently asking you about.
3. **The console output** will show some output when you run the program using the options in the toolbar (when available).

The toolbar includes a highlight button you can use to mark code selected in the editor, and an erase button to delete highlights.

In this study we will ask you to highlight parts of the code to assess your understanding of it. For example, variable declarations in TypeScript can be followed with a type declaration. We have highlighted one already here.

*For your first task, please highlight **all** the type annotations **that are not** `string`, and then click submit.*

Toolbar Highlight... Erase highlight...

```
1  var f: (x: number) => void = console.log
2  {
3    let int: number = 4
4    int<=2;
5  }
6  var str: string = "hello"
7  function double (x: number) : string {
8    return (x * 2).toString()
9  }
10
11 var len: number = double(20).length
12
```

Console output
>

Submit Highlights

Figure E.1: Tool tutorial, phase 1/5

Identifying the causes of runtime type errors with dynamic program slices

Tutorial (continuation)

Great Start! Now, if you make mistakes while adding annotations, you can use the erase button to remove an annotation at the cursor location.

For your second task, please remove all the highlights in this program, then click submit.

Toolbar Highlight... Erase highlight...

```
1  var messages: any = ["hello", "bye"]
2  var say: string = messages
3  console.log(say.toUpperCase())
4
```

Console output
>

Submit Highlights

Figure E.2: Tool tutorial, phase 2/5

Identifying the causes of runtime type errors with dynamic program slices

Tutorial (continuation)

Investigating changes to code

By design, TypeScript does not enforce types when you run a program.

You may have experienced that this allows type inconsistencies to sneak in code accepted by the TypeScript compiler.

Let's take a look at the following program, which is completely annotated and accepted by the TypeScript compiler. Click the Run button to see the output:

Toolbar Run... Highlight... Erase highlight...

```
1  var person: {id:string} = {id:'hello'}
2  var base_id: number = 1
3  function add_id (x: {}) : {id? : number } {
4    let ans : {id? : number} = x
5    ans.id = base_id++
6    return ans
7  }
8
9  var z = add_id(person)
10
11 console.log(person.id.toUpperCase())
```

Console output
>

I have followed the instructions of this task. Let me continue.

Identifying the causes of runtime type errors with dynamic program slices

Tutorial (continuation)

Investigating changes to code

By design, TypeScript does not enforce types when you run a program.

You may have experienced that this allows type inconsistencies to sneak in code accepted by the TypeScript compiler.

Let's take a look at the following program, which is completely annotated and accepted by the TypeScript compiler. Click the Run button to see the output:

Toolbar Run... Highlight... Erase highlight...

```
1  var person: {id:string} = {id:'hello'}
2  var base_id: number = 1
3  function add_id (x: {}) : {id? : number } {
4    let ans : {id? : number} = x
5    ans.id = base_id++
6    return ans
7  }
8
9  var z = add_id(person)
10
11 console.log(person.id.toUpperCase())
```

Console output
>ERROR: 11: person.id.toUppercase is not a function

I have followed the instructions of this task. Let me continue.

Figure E.3: Tool tutorial, phase 3/5

Identifying the causes of runtime type errors with dynamic program slices

Tutorial (continuation)

Investigating changes to code

We are investigating TypeSlicer, a tool that provides a runtime analysis of the types in a program.

TypeSlicer provides two features:

1. It tracks type information and checks for inconsistencies throughout program evaluation.
2. Once it detects a type contradiction, TypeSlicer backtracks evaluation and identifies the parts of the program that were not involved in the contradiction.

   The user interface is then able to **hide** parts of the code guaranteed to not be involved in the contradiction.

Let's revisit the same program we saw before, but this time we will activate TypeSlicer.

Note that the toolbar has a new element showing whether TypeSlicer is available or not. When it is available, it also has a toggle that lets you hide or show the code identified by the tool.

Click on Run again to see how the behaviour changes.

Toolbar | Run... | Highlight... | Erase highlight... | TypeSlicer is ON

```
1  var person: {id:string} = {id:'hello'}
2  var base_id: number = 1
3  function add_id (x: {}) : {id? : number } {
4    let ans : {id? : number} = x
5    ans.id = base_id++
6    return ans
7  }
8
9  var z = add_id(person)
10
11 console.log(person.id.toUpperCase())
```

Console output
>

I have followed the instructions of this task. Let me continue.

Identifying the causes of runtime type errors with dynamic program slices

Tutorial (continuation)

Investigating changes to code

We are investigating TypeSlicer, a tool that provides a runtime analysis of the types in a program.

TypeSlicer provides two features:

1. It tracks type information and checks for inconsistencies throughout program evaluation.
2. Once it detects a type contradiction, TypeSlicer backtracks evaluation and identifies the parts of the program that were not involved in the contradiction.

   The user interface is then able to **hide** parts of the code guaranteed to not be involved in the contradiction.

Let's revisit the same program we saw before, but this time we will activate TypeSlicer.

Note that the toolbar has a new element showing whether TypeSlicer is available or not. When it is available, it also has a toggle that lets you hide or show the code identified by the tool.

Click on Run again to see how the behaviour changes.

Toolbar | Run... | Highlight... | Erase highlight... | TypeSlicer is ON

```
1  var person: {id:string} = {id:'hello'}
2  [illegible]
3  function add_id (x: {}) [illegible] {
4    let ans : {id? : number} = x
5    [illegible]
6    [illegible]
7  }
8
9  [illegible] = add_id(person)
10
11 [illegible]
```

Console output
>**TypeSlicer Error: Line 4: Contradiction: Expected a {id?: number} and got a {id: string}**

I have followed the instructions of this task. Let me continue.

Figure E.4: Tool tutorial, phase 4/5

Identifying the causes of runtime type errors with dynamic program slices

Tutorial (continuation)

Investigating changes to code

Now that you have seen how TypeSlicer works, we can proceed to our main task.

In this study, we would like to know how **you** would change some programs to fix their type contradictions. We will ask you to run some programs, and then to **use the highlighting options in the toolbar** to let us know the parts of the code you think should be changed.

We will ask you to pick one of the possible fixes you can imagine for the code. If your particular fix requires more than one location to be changed, please highlight all of them.

To finish this tutorial, **assume you have decided that the proper change for this program is to alter the name of the `id` property in `person` to `name`.**

Please Run the program, then highlight the parts of the code that need to be changed to perform this fix. Then submit your fix proposal.

Toolbar Run... Highlight... Erase highlight... TypeSlicer is ON

```
1  var person: {id:string} = {id:'hello'}
2  var base_id: number = 1
3  function add_id (x: {}) : {id? : number } {
4    let ans : {id? : number} = x
5    ans.id = base_id++
6    return ans
7  }
8
9  var z = add_id(person)
10
11 console.log(person.id.toUpperCase())
```

Console output
>

Submit Highlights: These are the highlights requested in the instructions.

Identifying the causes of runtime type errors with dynamic program slices

Tutorial (continuation)

Investigating changes to code

Now that you have seen how TypeSlicer works, we can proceed to our main task.

In this study, we would like to know how **you** would change some programs to fix their type contradictions. We will ask you to run some programs, and then to **use the highlighting options in the toolbar** to let us know the parts of the code you think should be changed.

We will ask you to pick one of the possible fixes you can imagine for the code. If your particular fix requires more than one location to be changed, please highlight all of them.

To finish this tutorial, **assume you have decided that the proper change for this program is to alter the name of the `id` property in `person` to `name`.**

Please Run the program, then highlight the parts of the code that need to be changed to perform this fix. Then submit your fix proposal.

Toolbar Run... Highlight... Erase highlight... TypeSlicer is ON

```
1  var person: {id:string} = {id:'hello'}
2  var base_id: number = 1
3  function add_id (x: {}) : {id? : number } {
4    let ans : {id? : number} = x
5    ans.id = base_id++
6    return ans
7  }
8
9  var z = add_id(person)
10
11 console.log(person.id.toUpperCase())
```

Console output
>TypeSlicer Error: Line 4: Contradiction: Expected a {id?: number} and got a {id: string}

Submit Highlights: These are the highlights requested in the instructions.

Figure E.5: Tool tutorial, phase 5/5

Identifying the causes of runtime type errors with dynamic program slices

Task 1 of 6

All tasks share the same description, the only changes are the programs and whether the TypeSlicer tool is available to you.

Imagine you just joined a team developing a TypeScript application. This week you have been assigned to debug some failing tests in the application. The program compiles, but the test fails with a runtime error.

You do not yet have commit privileges to the repo, but you can contact James, the owner of this bug report. James has many meetings this week, so he would welcome any suggestions on how to fix this bug, even if some tests do not have an obvious solution or specification.

Your team's development environment includes TypeSlicer.

(1/3): Run the code. Think of a candidate fix for this error. What parts of the program need to be changed? Let us know **every part of the code that needs to be changed** using the highlighting options in the toolbar.

Toolbar Run... Erase highlight... TypeSlicer is ON

```
1  function map (f: (x: any) => any, l: Array<any>): Array<any> {
2    if (l.length === 0) {
3      return []
4    }
5    return [f(l[0])].concat(l.slice(1))
6  }
7
8  function example (list : Array<String>) : Array<Number> {
9    return map((x: String) => { return x.length }, list)
10 }
11
12 console.assert(
13   example(["a", "b", "c"]).every((x: Number) => x === 1),
14   "example must produce a list of 1's"
15 )
```

Console output
>

(2/3): **Write a change recommendation**. Please summarize a proposed fix for this bug: Imagine that what you write here will be sent to James and be attached to the bug report.

Write your change recommendation for James here

(3/3): Do you have any comments on the behaviour of this program? Was there anything surprising or unexpected for you?

Write your comments on the behaviour here

I'm finished with this task. Get me to the next task!

User study version 1.0 - last update Apr 14, 2023

Figure E.6: Task interface, before running a program (TypeSlicer available)

Identifying the causes of runtime type errors with dynamic program slices

Task 1 of 6

All tasks share the same description, the only changes are the programs and whether the TypeSlicer tool is available to you.

Imagine you just joined a team developing a TypeScript application. This week you have been assigned to debug some failing tests in the application. The program compiles, but the test fails with a runtime error.

You do not yet have commit privileges to the repo, but you can contact James, the owner of this bug report. James has many meetings this week, so he would welcome any suggestions on how to fix this bug, even if some tests do not have an obvious solution or specification.

Your team's development environment includes TypeSlicer.

(1/3): Run the code. Think of a candidate fix for this error. What parts of the program need to be changed? Let us know **every part of the code that needs to be changed** using the highlighting options in the toolbar.

Toolbar Run... Highlight... Erase highlight... TypeSlicer is ON

```
1  function map (, l: Array<>): Array<> {
2    if (l.length
3      return []
4    }
5    return [].concat(l.slice(1))
6  }
7
8  function example (list : Array<String>) : Array<Number> {
9    return map(, list)
10 }
11
12 console.assert(
13   example([, , ]).every(),
14   "example must produce a list of 1's"
15 )
```

Console output

```
>Error: Line 9:10 Expected map to
produce an Array<Number> but got [1,
"b", "c"]
```

(2/3): **Write a change recommendation**. Please summarize a proposed fix for this bug: Imagine that what you write here will be sent to James and be attached to the bug report.

Write your change recommendation for James here

(3/3): Do you have any comments on the behaviour of this program? Was there anything surprising or unexpected for you?

Write your comments on the behaviour here

I'm finished with this task. Get me to the next task!

User study version 1.0 - last update Apr 14, 2023

Figure E.7: Task interface, after running a program (TypeSlicer is on)

Identifying the causes of runtime type errors with dynamic program slices

Task 1 of 6

All tasks share the same description, the only changes are the programs and whether the TypeSlicer tool is available to you.

Imagine you just joined a team developing a TypeScript application. This week you have been assigned to debug some failing tests in the application. The program compiles, but the test fails with a runtime error.

You do not yet have commit privileges to the repo, but you can contact James, the owner of this bug report. James has many meetings this week, so he would welcome any suggestions on how to fix this bug, even if some tests do not have an obvious solution or specification.

Your team's development environment includes TypeSlicer.

(1/3): Run the code. Think of a candidate fix for this error. What parts of the program need to be changed? Let us know **every part of the code that needs to be changed** using the highlighting options in the toolbar.

Toolbar Run... Highlight... Erase highlight... TypeSlicer is OFF

```
1  function map (f: (x: any) => any, l: Array<any>): Array<any> {
2    if (l.length === 0) {
3      return []
4    }
5    return [f(l[0])].concat(l.slice(1))
6  }
7
8  function example (list : Array<String>) : Array<Number> {
9    return map((x: String) => { return x.length }, list)
10 }
11
12 console.assert(
13   example(["a", "b", "c"]).every((x: Number) => x === 1),
14   "example must produce a list of 1's"
15 )
```

Console output

```
>Error: Line 9:10 Expected map to produce an Array<Number> but got [1, "b", "c"]
```

(2/3): **Write a change recommendation**. Please summarize a proposed fix for this bug: Imagine that what you write here will be sent to James and be attached to the bug report.

Write your change recommendation for James here

(3/3): Do you have any comments on the behaviour of this program? Was there anything surprising or unexpected for you?

Write your comments on the behaviour here

I'm finished with this task. Get me to the next task!

User study version 1.0 - last update Apr 14, 2023

Figure E.8: Task interface, after running a program (TypeSlicer is off)

Identifying the causes of runtime type errors with dynamic program slices

Task 2 of 6

All tasks share the same description, the only changes are the programs and whether the TypeSlicer tool is available to you.

Imagine you just joined a team developing a TypeScript application. This week you have been assigned to debug some failing tests in the application. The program compiles, but the test fails with a runtime error.

You do not yet have commit privileges to the repo, but you can contact James, the owner of this bug report. James has many meetings this week, so he would welcome any suggestions on how to fix this bug, even if some tests do not have an obvious solution or specification.

Your team's development environment includes TypeSlicer. **For this task, the tool is unavailable.**

(1/3): Run the code. Think of a candidate fix for this error. What parts of the program need to be changed? Let us know **every part of the code that needs to be changed** using the highlighting options in the toolbar.

Toolbar Run... Highlight... Erase highlight...

```
1  function wrap (x: () => number): () => number {
2    return x
3  }
4
5  function wrap_again (x) : () => number {
6    return wrap(x)
7  }
8
9  let container = wrap_again( () => "200000000")
10
11 function main (f: () => number): bigint {
12   return BigInt( f() )
13 }
14
15 main(container)
```

Console output
>

(2/3): **Write a change recommendation**. Please summarize a proposed fix for this bug: Imagine that what you write here will be sent to James and be attached to the bug report.

Write your change recommendation for James here

(3/3): Do you have any comments on the behaviour of this program? Was there anything surprising or unexpected for you?

Write your comments on the behaviour here

You can continue to the next task here after 2 minutes

User study version 1.0 - last update Apr 14, 2023

Figure E.9: Task interface, before running a program (TypeSlicer not available)

Identifying the causes of runtime type errors with dynamic program slices

Task 2 of 6

All tasks share the same description, the only changes are the programs and whether the TypeSlicer tool is available to you.

Imagine you just joined a team developing a TypeScript application. This week you have been assigned to debug some failing tests in the application. The program compiles, but the test fails with a runtime error.

You do not yet have commit privileges to the repo, but you can contact James, the owner of this bug report. James has many meetings this week, so he would welcome any suggestions on how to fix this bug, even if some tests do not have an obvious solution or specification.

Your team's development environment includes TypeSlicer. **For this task, the tool is unavailable.**

(1/3): Run the code. Think of a candidate fix for this error. What parts of the program need to be changed? Let us know **every part of the code that needs to be changed** using the highlighting options in the toolbar.

Toolbar Run... Highlight... Erase highlight...

```
1  function wrap (x: () => number): () => number {
2    return x
3  }
4
5  function wrap_again (x) : () => number {
6    return wrap(x)
7  }
8
9  let container = wrap_again( () => "200000000")
10
11 function main (f: () => number): bigint {
12   return BigInt( f() )
13 }
14
15 main(container)
```

Console output

```
>Error: Line 12:10: Cannot convert
"200000000" to a BigInt.
```

(2/3): **Write a change recommendation**. Please summarize a proposed fix for this bug: Imagine that what you write here will be sent to James and be attached to the bug report.

Write your change recommendation for James here

(3/3): Do you have any comments on the behaviour of this program? Was there anything surprising or unexpected for you?

Write your comments on the behaviour here

I'm finished with this task. Get me to the next task!

User study version 1.0 - last update Apr 14, 2023

Figure E.10: Task interface, after running a program (TypeSlicer not available)

3. Please report how much you agree with the following statements: *(Each is presented with 5 options: "Strongly disagree", "Somewhat disagree", "Neither agree nor disagree", "Somewhat agree", and "Strongly agree")*
    - I think that I would like to use this tool frequently.
    - I found the tool unnecessarily complex.
    - I thought the tool was easy to use.
    - I think that I would need the support of a technical person to be able to use this tool.
    - I found the various functions in this tool were well integrated.
    - I thought there was too much inconsistency in this system.
    - I would imagine that most people would learn to use this tool very quickly.
    - I found the tool very cumbersome to use.
    - I felt very confident using the tool.
    - I needed to learn a lot of things before I could get going with this tool.