On The Design of a Gradual Dependently Typed Language for Programming

by

Joseph S. Eremondi

BSc. Honours Computer Science, University of Saskatchewan, 2014
BSc. Mathematics, University of Saskatchewan, 2014
MSc. Computing Science, Universiteit Utrecht, 2016

A THESIS 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)

March 2023

© Joseph S. Eremondi, 2023
The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

**On The Design of a Gradual Dependent Typingly Typed Language for Programming**

submitted by **Joseph S. Eremondi** 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 Bowman, Assistant Professor, Computer Science, UBC
Supervisory Committee Member

Éric Tanter, Professor, Computer Science Department, University of Chile
Supervisory Committee Member

Ivan Beschastnikh, Associate Professor, Computer Science, UBC
University Examiner

Sathish Gopalakrishnan, Associate Professor, Electrical and Computer Eng., UBC
University Examiner

Daniel Licata, Associate Professor, Computer Science, Wesleyan University
External Examiner
Abstract

Dependently typed programming languages provide a way to write programs, specifications, and correctness proofs using a single language. If a dependent type checker accepts a program, the programmer can be assured that it behaves according to the specification given in its types. However, dependently typed programming languages can be hard to use.

Gradual types provide a way to mix dynamically and statically typed code in a single language. Under this paradigm, programs may have imprecise types, causing certain type checks to be deferred to run time.

We build the theoretical foundations for combining gradual and dependent types in a programming language, with the aim of making dependent types easier to use. The differences between these two paradigms lead to inherent tensions when choosing the properties such a language should satisfy. Gradual typing’s effectful nature conflicts with the compile-time reductions of dependent type checking. Gradual run-time type comparisons clash with dependent types containing terms that bind variables. This dissertation identifies such tensions and proposes a design that finds balance between the conflicting goals.

Our contribution has three parts:

First, we present a foundational calculus for gradual dependent types, with functions, function types and universes. To ensure that type checking terminates, we reduce compile-time terms with approximate normalization, producing imprecise results when the available type information cannot guarantee termination. We use hereditary substitution to show that approximate normalization always terminates.

Second, we present a notion of propositional equality for gradual dependent
types. We devise a method of tracking run-time consistency information between imprecise equated terms, and introduce a composition operator in the language itself.

Third, we show that the first and second contributions can be combined, giving a language with approximate normalization that supports inductive types and propositional equality with dynamic consistency tracking. Since hereditary substitution does not scale to inductive types, we use a syntactic model to establish termination. The same technique is used to model non-terminating run-time semantics using guarded type theory, paving the road for mechanizing the metatheory of gradual dependent types.
Lay Summary

Programming is hard, and software with mistakes is frequently released. Dependently typed programming languages give a way for programmers to specify how programs should behave, and to write mathematical proofs that programs meet those specifications.

However, programming with dependently typed languages is hard. If a program passes a dependent type checker, the programmer can be confident it is correct, but to satisfy the type checker, they must accurately specify the behaviour of their program and mathematically prove that it fulfills the desired properties. As such, the barrier to entry is high for dependent types.

This dissertation develops the theory of *gradual dependent types*, a system that lets the programmer gradually introduce specifications and choose whether those specifications should be proved before the program runs or checked while it runs. Our work provides a balance between safety and flexibility, laying a foundation that makes dependent types easier to use.
Preface

The text of Chapter 1 draws from Eremondi, Tanter, and Garcia (2019) [58] of which I was the lead author. In particular, the motivating example with quicksort and length-indexed vectors is drawn from that work.

A version of Chapter 3 is published [J. Eremondi, E. Tanter, and R. Garcia. Approximate normalization for gradual dependent types. Proc. ACM Program. Lang., 3 (ICFP):88:1–88:30, July 2019. ISSN 2475-1421]. The use of compile-time evidence is novel to this dissertation, and corrects an error in the published work. I was the lead author, conceiving of the idea, developing the calculus, and constructing the proofs. R. Garcia and É. Tanter were supervisory authors and were involved throughout the research and manuscript preparation. É. Tanter was responsible for the normalization gradual guarantee.

The text of Chapter 4 draws from Eremondi, Garcia, and Tanter (2022) [60], and presents material originally published by Lennon-Bertrand, Maillard, Tabareau, and Tanter (2022) [87] under a license that permits abstracting with credit.

A version of Chapter 5 is published [J. Eremondi, R. Garcia, and E. Tanter. Propositional equality for gradual dependently typed programming. Proc. ACM Program. Lang., 6(ICFP), August 2022]. I was the lead author, inventing equality witnesses and constructing the language. R. Garcia and É. Tanter were supervisory authors and were involved throughout the research and manuscript preparation.

The remaining chapters are the original, unpublished work of J. Eremondi.
Table of Contents

Abstract ........................................ iii
Lay Summary ..................................... v
Preface ........................................... vi
Table of Contents ............................... vii
List of Figures ................................. xiii
List Of Abbreviations ............................ xv
Glossary .......................................... xvii
Acknowledgments ............................... xxi
Dedication ....................................... xxiv

1 Introduction ................................... 1
  1.1 Why Gradual Dependent Types? .......... 1
     1.1.1 The Best of Both Worlds: Gradual Types ... 2
     1.1.2 Extreme Static Typing: Dependent Types ... 3
     1.1.3 The Need for Gradual Dependent Types ... 4
  1.2 A Note on Typography ......................... 6
  1.3 A Motivating Example ......................... 7
     1.3.1 The Pain and Promise of Dependent Types ... 7
# Background and Related Work

2.1 Gradual Typing ............................................................... 17
  2.1.1 Imprecise Types ....................................................... 17
  2.1.2 Gradual Metatheory ............................................... 19
  2.1.3 Abstracting Gradual Typing .................................... 23

2.2 Dependent Types ............................................................. 27
  2.2.1 Essential Dependent Types ..................................... 28
  2.2.2 Inductive Families and Pattern Matching .................. 32
  2.2.3 Generic Representations of Inductive Types ............... 41
  2.2.4 Dependent Types: Implementation and Practical Details . 46
  2.2.5 Syntactic Models of Dependently Typed Languages ...... 50

2.3 Prior Work on Gradual Dependent Types .......................... 51
  2.3.1 Contracts and Dynamic Checking ............................. 52
  2.3.2 Gradual Type Theories .......................................... 53
  2.3.3 Dependently Typed Non-termination ......................... 54

# Foundations of Gradual Dependent Types

3.1 Introduction ............................................................... 56
  3.1.1 The unknown as a type index ................................. 60
  3.1.2 The unknown as a term at run time ......................... 60
  3.1.3 Managing effects from gradual typing ...................... 61
  3.1.4 Gradual Guarantees for GDTL ................................. 63
  3.1.5 Summary of Design Decisions ................................. 64

3.3 GDTL: A Static Dependently Typed Language .................... 65
  3.3.1 Syntax and Dynamic Semantics ............................... 65
  3.3.2 Comparing Types: Normal Forms ............................ 66
  3.3.3 Type checking and Normalization ............................ 67
  3.3.4 Hereditary Substitution ....................................... 70
  3.3.5 Properties of GDTL ............................................. 74
4.2 The Bidirectional CIC ............................................. 117
   4.2.1 Syntax .................................................. 117
   4.2.2 Typing and Semantics .................................. 118
4.3 GCIC: The Surface Language .................................. 121
4.4 CastCIC: The Cast Calculus .................................. 122
   4.4.1 Syntax, Typing and Reductions ...................... 122
   4.4.2 Elaboration .............................................. 124

5 Propositional Equality and Dynamic Consistency Tracking .... 129
   5.1 Gradual Propositional Equality ............................ 130
   5.2 Setting The Stage ........................................... 132
      5.2.1 Programming vs. Proving and the Gradual Guarantees .. 132
      5.2.2 Relationship to Existing Languages .................. 133
      5.2.3 A Motivating Example: Eagerly Enforcing Specifications . 134
      5.2.4 Lazily Enforcing Specs: Function Equalities and Extensionality .............................................. 139
   5.3 Propositional Equality ....................................... 143
      5.3.1 $GEQ$ Syntax and Typing ............................. 145
      5.3.2 $CastEq$ Syntax and Typing .......................... 145
      5.3.3 Cast Semantics ........................................ 146
      5.3.4 Semantics of Composition ............................. 149
   5.4 Parameterized Metatheory: Criteria for Precision and Consistency 153
      5.4.1 Stating the Gradual Guarantees ...................... 153
      5.4.2 Necessary Properties of Precision and Consistency .... 154
      5.4.3 Metatheory: Proving Safety and the Gradual Guarantees . 157
   5.5 Consistency and Precision .................................. 159
      5.5.1 Precision and Consistency in $GCIC$ .................. 160
      5.5.2 Precision and Consistency for $GEQ$ .................. 161
      5.5.3 Fulflling The Criteria ............................... 165
   5.6 Discussion .................................................. 167
      5.6.1 Extensions Enabled by Equality ...................... 167
      5.6.2 Future Work ......................................... 170
      5.6.3 Related Work ....................................... 172
# List of Figures

| Figure 3.1 | SDTL: Syntax and Semantics | 66 |
| Figure 3.2 | SDTL: Normal and Neutral Forms | 68 |
| Figure 3.3 | SDTL: Type Synthesis and Checking | 69 |
| Figure 3.4 | SDTL: Normalization | 71 |
| Figure 3.5 | SDTL: Auxiliary Judgments for Normal Forms | 72 |
| Figure 3.6 | SDTL: Hereditary Substitution | 73 |
| Figure 3.7 | GDTL: Terms and Normal Forms | 76 |
| Figure 3.8 | GDTL: Concretization and Consistency | 78 |
| Figure 3.9 | GDTL: Abstraction and Lifted Functions | 80 |
| Figure 3.10 | GDTL: Type Checking and Synthesis | 82 |
| Figure 3.11 | GDTL: Well-Formedness Rules | 83 |
| Figure 3.12 | Type Derivation for head of nil | 85 |
| Figure 3.13 | Approximate Neutral Substitution, Key Rules | 89 |
| Figure 3.14 | GDTL: Approximate Normal Substitution | 90 |
| Figure 3.15 | GDTL: $\eta$-expansion | 92 |
| Figure 3.16 | GDTL: Approximate Normalization | 95 |
| Figure 3.17 | GDTL: Meta-level Casting | 97 |
| Figure 3.18 | GDTL: Meta-level Casting to the unknown type | 98 |
| Figure 3.19 | GDTL: Meta-level Casting from the unknown type | 99 |
| Figure 3.20 | GDTL: Composing Normal Forms | 99 |
| Figure 3.21 | Evidence Term Syntax and Typing | 102 |
| Figure 3.22 | GDTL: Dynamic Semantics | 105 |
| Figure 3.23 | Gradual Precision of Normals, Neutrals and Environments | 109 |
| Figure 4.1 | BCIC: Syntax | 117 |
| Figure 4.2 | BCIC: Semantics | 119 |
| Figure 4.3 | Bidirectional cic: Syntax, Typing and Semantics | 120 |
| Figure 4.4 | GCIC: Syntax and Typing Lemmas | 121 |
| Figure 4.5 | CASTCIC: Typing and New Syntax | 123 |
| Figure 4.6 | CASTCIC: Propagation Reduction Rules | 124 |
| Figure 4.7 | CASTCIC: Cast Reduction Rules | 125 |
| Figure 4.8 | Elaboration from GCIC to CASTCIC | 126 |
| Figure 4.9 | Elaboration from GCIC to CASTCIC (ctd.) | 127 |
| Figure 4.10 | CASTCIC Consistency and Consistent Convertibility | 128 |
| Figure 5.1 | Sorting Fixed-Length Lists | 136 |
| Figure 5.2 | GEq: Syntax and Typing Lemmas | 146 |
| Figure 5.3 | CASTEq: Syntax, Typing and Elaboration Rules | 147 |
| Figure 5.4 | CASTEq: Reductions for Casts and Equality | 148 |
| Figure 5.5 | CASTEq: Semantics for Composition | 150 |
| Figure 5.6 | Positive Types | 151 |
| Figure 5.7 | Telescope Composition | 152 |
| Figure 5.8 | Structural Precision: Generating and Cast Rules | 161 |
| Figure 5.9 | Precision: Diagonal Rules for GCIC forms | 162 |
| Figure 5.10 | Definitional Precision | 163 |
| Figure 5.11 | Precision for Propositional Equality and Composition | 164 |
| Figure 5.12 | Precision Modulo Conversion | 164 |
| Figure 5.13 | Consistency for Propositional Equality and Composition | 164 |
| Figure 6.1 | Ordering on Sizes | 218 |
| Figure 6.2 | Model of Composition | 221 |
| Figure 6.3 | Model of Composition for Codes | 222 |
| Figure 6.4 | Model of Casting | 224 |
| Figure 6.5 | Model of Casting (ctd.) | 225 |
| Figure 6.6 | Casting Functions To and From the Germ | 226 |
| Figure 6.7 | The Unknown Term for Codes | 228 |
| Figure 6.8 | Syntactic Model of *GEq: Summary | 231 |
List Of Abbreviations

\textbf{AGT}  Abstracting Gradual Typing.

\textbf{BCIC} the Bidirectional Calculus of Inductive Constructions.

\textbf{CASTCIC} Cast calculus for the Gradual Calculus of Inductive Constructions.

\textbf{CASTEq} Cast calculus with propositional \textit{Equality}.

\textbf{cc}_\omega \textnormal{ the Calculus of Constructions with a predicative universe hierarchy.}

\textbf{CIC} the Calculus of Inductive Constructions.


\textbf{FFI} Foreign function interface.

\textbf{GCIC} the Gradual Calculus of Inductive Constructions.

\textbf{GDTL} Gradual \textit{Dependently Typed Language}.

\textbf{GEQ} Gradual language with propositional \textit{Equality}.

\textbf{GEQ} Guarded Gradual language with propositional \textit{Equality} and decidable type checking.

\textbf{GRIP} a “reasonably gradual” type theory.

\textbf{GTT} Guarded Type Theory.
**MLTT**  Martin-Löf Type Theory.

**OTT**  Observational Type Theory.

**REPL**  Read Eval Print Loop.

**SDTL**  Static Dependently Typed Language.

**STLC**  the Simply Typed Lambda Calculus.

**TCTT**  Ticked Cubical Type Theory.

**UIP**  Uniqueness of Identity Proofs.
Glossary

**approximate normalization** A method of reducing programs to a normal form for gradual types. Whenever type information is not precise enough to guarantee that a term can halt, an imprecise term is produced as a result. Approximate normalization ensures that type checking terminates.

**axiom K** The principle that reflexivity is the only proof that a term is equal to itself. Equivalent to UIP.

**canonicity** The property that, for every type $T$, every closed term of type $T$ can either be reduced, or is constructed with a type former for $T$.

**cast calculus** An intermediate language for gradual typing, where imprecise types are allowed, but all conversions between types are explicitly marked by a casting operation.

**closed type theory** A type theory with a fixed set of type formers, where new inductive types cannot be declared.

**consistent transitivity** For a given relation, the operation in Abstracting Gradual Typing where, given evidence that $T_1$ and $T_2$ are related, and evidence that $T_2$ and $T_3$ are related, the two pieces of evidence are composed to determine if $T_1$ and $T_3$ are known to be related. Collapses to the precision-meet when the relation is the gradual lifting of equality.

**container** See W-type.

**cumulativity** The principle that a term with type $Type_\ell$ can also be typed at $Type_{\ell+1}$.
Curry-Howard correspondence A correspondence between constructive logic and programming languages, where propositions correspond to types, programs correspond to proofs, and inhabited types correspond to theorems.

definitional equality The notion of equality used to compare terms during dependent type checking. Often it is defined as syntactic $\alpha$-equivalence of normal forms.

dependent pair A dependent type whose elements contain two values, where the type of the second may refer to the value of the first.

dynamic gradual guarantee The property that reducing a program’s precision causes no new dynamic type errors, and that other than errors, changing a program’s precision does not alter its behaviour.

function extensionality The principle (or axiom) that two functions are propositionally equal if they produce the equal outputs for all inputs.

functor In this dissertation: the function from $\text{Type}$ to $\text{Type}$ of which an inductive type is a fixed-point.

gradual guarantees See static gradual guarantee and dynamic gradual guarantee.

graduality The semantic notion of the gradual guarantee given by New and Ahmed [105], stating that precision-related terms produce precision-related results under any context. Used by Lennon-Bertrand et al. [87] to mean embedding-projection pair, since they are typically how graduality is established.

inductive type A type $T$ defined in terms of some number of constructors, each of which contains fields, some of which may be of type $T$.

motive A function producing a type, given as an argument to the eliminator for an inductive data type. If the motive is $P$, eliminating $t$ produces a value of type $P \, t$. 

xviii
normalization gradual guarantee  The property that all normalizations during
type checking obey the dynamic gradual guarantee.

observational equivalence  Two programs are observationally equivalent if, un-
der any observable context, they produce the same results. Common no-
tions of observation include whether two programs have the same halting
behavior under all contexts, or whether two programs produce the same
result in all boolean-typed contexts.

open type theory  A type theory which does not have a fixed set of type formers,
but allows for the declaration of new inductive types and their constructors.

proof terms  A term in a dependently typed program whose purpose is not to
specify behavior, but to establish that another piece of code fulfills some
property.

propositional equality  The internalization of definitional equality as a type in a
dependently typed programming language. For any two terms of some type,
the propositional equality type contains all proofs that those two terms are
equal, if any such proofs exist.

static gradual guarantee  The property that reducing a program’s precision causes
no new static type errors.

strictly positive  A function from types to types is called strictly positive if its
parameter never occurs to the left of a function arrow. An inductive defi-
tion is strictly positive if it is the fixed-point of a strictly positive function,
that is, it has no self-reference to the left of an arrow.

syntactic model  A method of defining semantics for a language by giving a
translation to another language, usually a logically-consistent dependently
typed language.

transport  A function that converts between two types, given a proof that those
types are equal.
**unknown type**  The least precise type in gradual typing. A term of the unknown type can be used in a context expecting a value of any type.

**W-type**  A generic method of defining inductive types, given by specifying a command type $C$ that specifies the constructor and values for non-recursive fields, and a response type $R$ for each command value, roughly denoting "how many" recursive references should accompany that command. The command and response together are called a container. The elements of $W C R$ consist of a command value $c : C$ and a function from $R c$ to $W C R$. 
Acknowledgments

I am grateful to NSERC for their financial support through the CGS-D scholarship. I am thankful to everyone who has helped me on this journey. In particular, I wish to thank:

• Ron Garcia: When I arrived at UBC, I thought studying programming languages meant using Haskell, that writing was about using big words to show how smart you were, and that I could treat conference talks like debate speeches, speaking quickly and angrily. In my time as your student, you have taught me so much, and you have done so with a gentleness and flexibility, without which I would have buckled under the stresses of a PhD. You have gone out of your way to ensure my studies were compatible with being a parent, and for some reason you put up with my last-minute deadline pushes and typo-ridden drafts. Thank you for being my mentor and guide.

• Éric Tanter: from the time Ron gave you my half-cooked error-ridden draft of our ICFP19 paper, you have been instrumental in helping me refine my ideas into something worth sharing, guiding me through existing literature and finding the interesting questions in my research.

• William Bowman: Whether as a senior student introducing me to Cur in Nara at the beginning of my PhD, or as a professor teaching me compiler theory, you have been a friend and guide throughout this process. One day, we will bring Ron to the light of dependent types.

• Daniel Licata, Mieszko Lis, Ivan Beschastnikh, and Sathish Gopalakrishnan:
Thank you for agreeing to be on my examination committee, for reviewing my dissertation, for your helpful suggestions and ideas for future work, and for keeping me on my toes with tough but fair questions during my defence.

- **Wouter Swierstra:** for your flexibility in helping me complete my Master’s thesis, and for teaching me how to collaborate remotely, a skill that proved very useful in 2020. Without you, I would not have begun my PhD, let alone finished it.

- **Dominic Orchard:** In 2012 I attended the SPLASH to compete in the undergraduate student research competition. I had never attended a conference before, and I was scared, shy and lonely. You talked to me and invited me out with your labmates, and visited with me throughout the conference. Your kindness was my introduction to the warmth found in throughout the programming languages community, and was part of why I ultimately wanted to be part of that community.

- **Jana Dunfield and Alex Summers:** through reading group and the SPL, you have both been valuable mentors. Thank you for all the programming language ideas, old and new, that you have introduced me to.

- **Felipe, Nico, Rui, Paulette, Jonathan, Adam, and all my other SPLabmates, classmates, reading group members and UBC friends:** From helping each other with our research, to keeping the mood light in the lab, to suffering through impenetrable category theory papers together, I am thankful for the fellowship we shared at UBC.

- **Andrej Bauer, András Kovács, Conor McBride, Daniel Gratzer, Jonathan Sterling, Amélia Liao, Paolo Giarrusso, Guillaume Allais, Jesper Cockx, Neel Krishnaswami, Andrea Vezzosi, and everyone else who answered my questions online:** you have provided me with so much knowledge, and you helped create a sense of community when I was stuck at home during the pandemic. Thank you.

- **My Dalmeny family:** You have supported me and my family throughout this entire process, financially, emotionally, and with your time. From driving
to Vancouver to help us move to the vast amount of child care you gave in my last year, I absolutely could not have finished my PhD without your support.

- **My Nanaimo family:** I am so grateful to have had family with me in British Columbia. From game nights, to baby sitting, to saving us during two family emergencies, you have been true friends to us. I love you, and I am so proud of how you have grown and what your family has become.

- **My parents:** You have encouraged my learning throughout my whole life, from reading to me about dinosaurs, to learning the world capitals with me, to proofreading high-school essays, to listening to whatever I was rambling on about at a given time. You have taught me, supported me, and believed in me, and loved me unconditionally. Thank you.

- **My wife, Joanne:** You have supported me unwaveringly through this long process, in so many ways. You have endured so much with me and sacrificed so much for us to reach this point. I cannot imagine what my life would be if I did not have you in it. I love you, and I could not have done this without you.
Dedication

For B and K, my reasons to go on.
Chapter 1

Introduction

1.1 Why Gradual Dependent Types?

Writing correct software is hard. As software becomes ubiquitous in society, the costs of incorrect software increase. The harms caused by programmer error include the loss of millions of dollars in currency exchanges [45], widespread corruption of analysis in genomics research [10], and even airplane crashes [148].

Many tools have been developed to prevent errors in programs, but the programmer’s first line of defence against bugs is the programming language. When given tools for abstraction, the programmer can design large scale programs without needing to reason about the byte-level behaviour of their program. Likewise, repeated behaviour can be separated into a function or method, letting the programmer reason about the behaviour of parts of the code without having to mentally model the entire program at once.

Two main paradigms have arisen in programming languages. *Statically typed* programming languages require each expression in a language to have a type, which can describe properties such as the possible forms that data can take, information flow, exceptions that can be raised, etc. Likewise, operations are assigned types to determine what expressions constitute valid arguments. These types may be specified by the programmer, or they may be inferred by a type checker, but they must be known before the program runs. A type checker verifies that code is well typed, and hence describes a valid program, before running or compiling it.
Dynamically typed languages impose no such restriction that types must be statically known, and programs that provide invalid operands to an operation may crash with a run-time error.

Static and dynamic checking have their own tradeoffs. Static checking can prevent a large class of bugs, and provide guidance to the programmer about what values are valid arguments to a function or operation. However, when writing a function, the programmer must know the types of its inputs and outputs before it can be tested. On the other hand, dynamic checking leave the possibility of run-time errors, but allow for easy code re-use and rapid prototyping.

1.1.1 The Best of Both Worlds: Gradual Types

Static and dynamic checking each have unique advantages, but their benefits are not mutually exclusive. Gradual typing [132] provides a principled approach to mixing statically and dynamically typed code. In a gradual language, the programmer can annotate some expressions with the unknown type \( ? \). Terms of type \( ? \) can be used in any context and built from values of any type. The key feature of (sound) gradual typing is that when typing information is not known statically, dynamic checks are automatically inserted into a program, ensuring that any run-time type errors lead to immediate program termination, rather than unsafe memory access or other undesired behaviour that might be exploitable by an attacker, or might inadvertently cause data loss or other damage.

Moreover, fully gradual languages support migration from dynamic enforcement of type invariants to static enforcement. The gradual guarantees of Siek et al. [136] state that decreasing the precision of a program never introduces new static or dynamic type errors. These guarantees ensure that, when an error is encountered, the problem is never too few types, but that some statics types ascribed to parts of the program are fundamentally incompatible. With the gradual guarantees, if a program is statically typeable, then that program is gradually typeable at every step between fully dynamic and fully static.
1.1.2 Extreme Static Typing: Dependent Types

Conventional static and dynamic typing are not capable of eliminating all bugs from code. Static checks can prevent dynamic type errors, and dynamic checks can ensure such errors never cause unsafe behaviour. However, correctness bugs can still cause problems in programs. Even if a program runs successfully, it might not behave in the way the programmer expects. Testing is not sufficient to guarantee correctness: if a program has infinitely many possible inputs, then it is impossible for testing alone to ensure that the program produces the desired output for each input.

However, dependently typed languages can guarantee the correctness of a program, at least up to the specification given in its type. Types may refer to (depend on) terms, which makes them expressive enough to write complete program specifications. For instance, the type of vectors is indexed by a number denoting its length. The terms occurring within types need not be values: types can depend on yet-to-be-evaluated expressions. Moreover, dependent types are first-class language constructs: in addition to containing computation, types can be the input to computation or the result of computation.

Dependent types are deeply connected to logic: Through a connection known as the Curry-Howard correspondence, each type corresponds to a logical proposition, and a term of that type denotes a proof that said proposition holds. This correspondence allows for programmers to write their programs, the correctness specifications for their programs, and the proof that the programs meet those specifications, using a single, unified language.

Dependent types can express invariants that are more fine-grained than non-dependent static type systems, such as ML or Haskell-style types, which we henceforth describe as simply-typed. Dependent type checkers can reject some programs accepted by simply-typed languages, such as those that call a function but violate said function’s precondition. Conversely, the ability of a function’s return type to depend on an argument’s value means that some idioms from dynamically typed languages can be expressed with static dependent types. There are programs that have no type in simply-typed languages, that can be assigned a type in a dependently typed language. Hence, with dependent types, developers
can write highly flexible, reliable code.

1.1.3 The Need for Gradual Dependent Types

Though dependently typed languages give the power to ensure that software is correct, they can be difficult to use. One can be confident that a program accepted by a dependent type checker meets the specification given by its types, but actually producing such a program and specification is hard. When verifying a C compiler using dependent types, Leroy [88] notes that the verification code is approximately six times larger than the program being verified. Ringer et al. [126] observe that most dependently typed developments are completed by small teams of experts, and that these experts often work closely with, or are themselves, developers of dependently typed languages.

The high barrier to entry for dependent types leads to a vicious cycle. The rigid type discipline and the need to manually write proofs means that dependently typed languages are difficult to learn, so few people use them for programming. Because few people use dependent types, the libraries and tooling for dependently typed languages are lacking. The dearth of libraries leads to a negative feedback loop: few libraries have been written, meaning that few people use dependent types, meaning that few people write libraries, etc.

Lowering the barrier to entry for dependent types disrupts this vicious cycle: if dependent types are easier to use, more people will use them, so more people will write tools and libraries for them, which in turn makes them easier to use, etc. Several efforts to improve the usability of dependent types are ongoing, including improved error messages [57], proof reusability [127], and metaprogramming tools to automate the construction of proof terms [28, 32].

However, among the remaining barriers is the difficulty of migrating code from non-dependent to dependent types, and the difficulty of constructing the proofs necessary for programs to type check. Migrating code from languages like Scheme, Haskell or ML is often not straightforward, and while these languages can be accessed via a Foreign function interface (FFI), doing so often violates the safety guarantees provided by dependent types. For instance, passing an empty list from OCaml to the dependently typed language Coq can lead to a null-pointer
dereference [42].

To alleviate these difficulties, we propose integrating gradual types with dependent types. This would provide multiple advantages.

1.1.3.1 Migration

Gradual types provide a smooth path for migrating from dynamically typed languages to those with static types. At each stage of the migration, the whole program can be run, even when static type information is missing, with dynamic checks performed to ensure safety. Gradual typing is particularly helpful with migrating to dependent types: in many places, standard dependently typed functions contain terms (called indices), or demand construction of a proof that some property holds. By expanding the notion of the unknown type to also include terms, gradual dependent types let programs run even when parts of type indices or proofs are missing.

1.1.3.2 Interoperability

Another use of gradual dependent types is to ensure a safe interface between dependently typed and non dependently typed code (as opposed to, say, an FFI that does not perform dynamic type checks). In such a system, typing guarantees are still ensured dynamically, and gradual dependent types enable enforcement of both basic types and sophisticated function preconditions across the boundary between languages.

1.1.3.3 Aiding Development

Several dependently typed languages are oriented around typed holes: while building a proof, the user may omit parts, and the editor informs them of type the omitted part (called the goal) should have, along with what premises (i.e. in-scope variables) they can use to construct a value of the goal type. However, terms with holes cannot be safely compiled, and while tools to evaluate terms with holes exist, they usually get stuck when a hole is reached, causing static type checks to be omitted without dynamic checks to replace them.
Gradual dependent types allow for hole-based development where evaluation does not get stuck on holes. Static checks are not omitted but are instead converted to dynamic checks. This provides better support for dynamic development tools, such as a Read Eval Print Loop (repl) or a test suite. We conjecture that information from dynamic runs of a program could even be useful in the automated construction of correctness proofs. Moreover, structure editors like Hazel [110, 111] allow for user-friendly editing in which every reachable editor state is a runnable program. The theory of such editors is based on gradual types, so gradual dependent types could facilitate the development of structured editors for dependently typed languages.

This dissertation lays the foundation for these goals. Our vision for gradual dependent types is to have a type system that provides a unified interface for specifying program properties, whether they are to be checked statically or dynamically. Every programmer has written code like `error('TODO write later')`, and we seek to provide foundations for language-level support to dynamically track the safety of such code. We do not envision gradual dependently typed code being widely used in production, but see it as a stepping stone on the path to releasing fully static dependently typed software, with the hope that it can help surpass some of the roadblocks that prevent dependently typed languages from being used at all. We do not yet have empirical evidence for the usefulness of our techniques, but we cannot test usefulness without an implementation, and we cannot implement gradual dependent types without adequate foundations.

1.2 A Note on Typography

Throughout this dissertation, we routinely discuss terms in a static language, along with their counterparts in a gradual language. For clarity, we write static dependently typed terms using red sans-serif font, and gradual surface terms using green, italic serif font. Terms from intermediate languages, such as a gradual cast calculus or evidence language, are written using blue, bold serif font.

Though this thesis can be read on paper, the digital version contains several enhancements that aid with reading. In particular, uses of acronyms and the first
uses of glossary terms link to their respective definitions. Likewise, references to sections are linked to section headers, inference rules are linked to their definitions, and citations are linked to their bibliography entries. Clicking these links can be used to navigate from a use to a definition, and in a sufficiently modern PDF viewer, such as Evince, hovering over such a link displays the respective definition in a pop-up frame.

1.3 A Motivating Example

In this section, we give a preview of our vision for gradual dependent types, along with an example of why we might want them. The example is adapted from our previous work [58].

1.3.1 The Pain and Promise of Dependent Types

1.3.1.1 Length-indexed Vectors

We begin with a classic example of dependently typed code: length-indexed vectors. We use vectors in examples throughout this dissertation, as they provide a minimal example of how dependent types can add safety to the typically unsafe operation of accessing the \( n \)th component of a list of unknown length.

Consider the type of vectors:

\[
\text{Vec} : \text{Type} \rightarrow \mathbb{N} \rightarrow \text{Type}
\]

For any type \( T \) and natural number \( n \), \( \text{Vec} T n \) is the type of vectors containing exactly \( n \) elements of type \( T \). \( \text{Vec} \) describes a family of types, and for each type \( T \) and number \( n \), \( \text{Vec} T n \) is a distinct type. The type \( \text{Vec} \) is generic not only over \( T : \text{Type} \) (its element type), but over \( n \) taken from \( \mathbb{N} \), the type of natural numbers. The type depends on a value, thus the name dependent type.

The \( \text{Vec} \) type has two constructors:

\[
\begin{align*}
\text{Nil} : & \quad \text{Vec} X 0 \\
\text{Cons} : & \quad X \rightarrow \text{Vec} X n \rightarrow \text{Vec} X (n + 1)
\end{align*}
\]
The constructor `Nil` creates a vector with length 0, while `Cons` takes a vector of length `n` and adds an element to produce one of length `n + 1`. In both cases, these length constraints are reflected in the constructor types: each has a return type of the form `Vec T m`, but the precise value of `m` differs between the two constructors, and in the case of `Cons`, `m` actually depends on the value of a previous argument’s length. We say that `m` is a `type index`, since the choice of `m` constrains which vectors can inhabit `Vec T m`.

Including the length of the vector in the type ensures that operations that are typically partial can only receive values for which they produce results. For instance, one can write a well typed function that yields the first element of a vector, using the type:

\[
\text{head} : (X : \text{Type}) \rightarrow (n : \mathbb{N}) \rightarrow \text{Vec } X \,(n + 1) \rightarrow X
\]

Since `head` takes a vector of non-zero length, in a static dependently typed language, it can never receive an empty vector. If the type checker accepts a call to `head`, then the programmer knows such a call will not raise an error. The downside of this strong guarantee is that `head` can only be called with vectors whose length is statically known to be non-zero, or the programmer must manually prove that the length cannot be zero. This burden makes it difficult to migrate code from languages with simpler types, and difficult for newcomers to rapidly prototype their algorithms.

### 1.3.1.2 Migrating Quicksort

To see the difficulty of using vectors, consider the task of migrating a simple sorting algorithm into a dependently typed language. Consider the non-dependently typed quicksort, written in pseudo-Haskell:

```haskell
sort : List \mathbb{N} \rightarrow List \mathbb{N}
sort Nil = Nil
sort (Cons h t) = (sort (filter (< h) t)) + (singleton h) + (sort (filter(\geq h) t))
```

Now consider the task of migrating this definition to a dependently typed lan-
guage. One benefit of using dependent types is that we can capture the invariant that \texttt{sort} preserves the size of the list by giving it the type:

\[
\texttt{sort} : \texttt{Vec} \, \texttt{N} \, n \rightarrow \texttt{Vec} \, \texttt{N} \, n
\]

The helper concatenation function can be assigned the type

\[
\texttt{+} : \texttt{Vec} \, \texttt{N} \, m \rightarrow \texttt{Vec} \, \texttt{N} \, n \rightarrow \texttt{Vec} \, \texttt{N} \, (m + n)
\]

However, assigning \texttt{filter} a type is more tricky, as we explain below, as is migrating the rest of the function:

\[
\texttt{sort} \, \texttt{Nil} = \texttt{Nil} \\
\texttt{sort} \, (\texttt{Cons} \, h \, t) = \texttt{transport} \, ????

((\texttt{sort} \, (\texttt{filter} \, (<) \, h) \, t)) + + (\texttt{singleton} \, h) + + (\texttt{sort} \, (\texttt{filter} \, (\geq) \, h) \, t))
\]

In order for this to be well typed, in the \texttt{Cons} case, the lengths of (sort (filter (<) h) t), singleton h and (sort (filter (≥) h) t)) must sum to the length of Cons h t. However, 1 + length (sort (filter (<) h) t) + length (sort (filter (≥) h) t)) is not syntactically identical to 1 + length t. Instead, we must use a \texttt{transport} function, which consumes a proof that the two types are equal and converts from one type to the other. We use ??? to denote the hole that the programmer must fill with such a proof.

A bigger issue for the migration is determining what the type of \texttt{filter} should be for vectors. It consumes a vector of type Vec T n and a predicate of type T \rightarrow \texttt{B} (where \texttt{B} is the type of booleans), but what should its return type be? It should have the form Vec T ???, but we cannot statically know the length without knowing exactly how many elements of the vector satisfy the predicate.

A final problem is that \texttt{sort} calls itself recursively on the result of \texttt{filter}. To ensure decidable type checking, most dependently typed languages require that recursive calls be made on structurally smaller arguments, or that the programmer manually provide some metric which is strictly decreasing for each recursive
call.\footnote{This requirement will cause us much grief in Chapter 6.} Because the recursive calls are made on the results of \texttt{filter}, they are not structurally smaller than the original list, and for the type checker to accept \texttt{sort}, the programmer must provide proof that each recursive call is made on a strictly shorter list.

It should be clear now that migrating a simple function is not necessarily a simple process. All of the above difficulties can be overcome: it is certainly possible to write a proof that the result has the same length as the input, though doing so involves lemmas about the length of concatenating lists and the associativity of addition. Likewise, \texttt{filter} can be rewritten to return a \emph{dependent pair} containing the number of elements satisfying the predicate and a vector of that length, and one can write a proof that filtering a list with a predicate and that predicate’s negation produces two lists whose lengths sum to that of the original list.

The issue is not that the above tasks are impossible, but that they are difficult. If a newcomer to dependent types tries to migrate some of their code, they are immediately faced with tasks that are not beginner-friendly. Even if they use holes to skip the parts they get stuck on, they cannot compile their code until those holes are filled. Alternately, one could use simply-typed lists in the dependently typed language, but this would require duplicating every vector function for lists. Existing solutions, like ornaments [41], require the standard libraries of dependently typed languages to be redesigned under a totally different programming paradigm.

\subsection*{1.3.1.3 Envisioning A Gradual Solution}

Gradual dependent types can ease the migration of the \texttt{sort} function. For \texttt{filter}, instead of using dependent pairs, one can write a \texttt{filter} function that returns a value of type:

\begin{equation}
\text{Vec } T ?
\end{equation}

That is, it returns a vector with an imprecise type. Likewise, the programmer can write \texttt{?} to give an imprecise proof to \texttt{transport} that the two lists have the same length, and similarly use an imprecise proof to show that recursive calls to \texttt{sort} are made on smaller lists. Unlike with holes, the programmer is free to run their
program or move on to developing different modules that use \texttt{sort}.

The final gradual sort is as follows:

\[
\text{sort } \text{Nil} = \text{Nil} \\
\text{sort } \text{(Cons } h \text{ } t) = \text{transport } ? \\
\quad (\text{(sort } \text{filter } (< h) t)) + + (\text{singleton } h) + + (\text{sort } \text{filter } (\geq h) t))
\]

This dissertation is devoted to designing safe, principled language mechanisms that support the above techniques for migrating code. In the following section, we outline the major design decisions and challenges that come with designing such a language.

1.3.2 Challenges and Solutions with Gradual Dependent Types

Integrating gradual and dependent types is non-trivial. Though techniques exist for systematically introducing gradual types into a static language \cite{33, 67}, they cannot handle types that depend on terms. We identify the primary challenges of integrating gradual and dependent types, and summarize the solutions presented in the rest of this dissertation.

1.3.2.1 The Unknown Type Is Not Enough

On its own, the unknown type \texttt{?} from non-dependent gradual types is not expressive enough for dependent types. One could write a filter function that returns type \texttt{?}, but that return type is too permissive: we know that \texttt{filter} should return a vector, and that the vector’s element type should be the same as the input to \texttt{filter}. The program might perform unnecessary run-time checks to ensure that the result is a list, and if the programmer were to edit the definition of \texttt{filter}, they could accidentally modify it to return a non-list value.

The solution is to introduce imprecision more judiciously, as in \texttt{Vec N ?}: the type of vectors of numbers with unknown length. But the length is a term (a natural number), not a type. So to be sufficiently expressive, gradual dependent types need a notion of imprecise term, which does not exist for non-dependent gradual types.
1.3.2.2 Proofs Can Persist at Run Time

Dependently typed programs often require *proof terms*, which establish that two expressions are equal or that some other precondition holds. Holes can be used to defer the construction of proofs, but programs containing these holes typically cannot be fully compiled or run, presenting a serious barrier to development. These proof terms are parts of the program, and may contain data or affect the run-time behaviour of the program in any number of ways.

To seamlessly blend untyped and dependently typed code, programmers should be able to omit proofs, yet still have their code type check and run. Moreover, whatever invariants the omitted proofs were supposed to be establishing should be checked at run time, to the extent that doing so is decidable, so that the programmer is alerted when a program violates its specification.

Our proposed solution is to let \( ? \) be used as an expression at run time, and to give semantics for how computing with it should behave. Giving \( ? \) dynamic semantics is an extension of the previous goal, since such a semantics lets \( ? \) be used in place of any term, not just type indices. Then, the user can omit proofs and still run their programs.

For the *sort* example, \( \text{transport} ??? \) could be replaced with \( \text{transport} ? \), which would result in a complete but imprecise program. When executed, the proof imprecision should induce a run-time check that the return vector actually has the same length as the input.

1.3.2.3 Gradual Typing Introduces Effects

Adding \( ? \) to types introduces two computational effects. The ability to type self-applications enables the construction of well typed fixed-point combinators. Thus, programs may run forever. Likewise, the ability to write imprecise types introduces the possibility of type errors that may be encountered while evaluating expressions.

These effects are troubling because, to compare types, dependent type checking must often evaluate code, sometimes under binders. Since the return type of a dependently typed function may depend on the value given, we must evaluate code at compile time to compute these return types. So both effects can manifest
during type checking.

For non-termination, our solution is to devise a separate notion of compile-time and run-time evaluation, ensuring that type checking terminates and properly handle errors. The compile-time notion approximates terms to ? whenever it cannot guarantee their termination, while run-time evaluation is precise but may run forever. A large portion of this dissertation is dedicated to establishing that compile-time normalization is decidable, since proving termination is non-trivial, particularly for a language supporting inductive types and propositional equality.

To account for dynamic type errors during type checking, we present two approaches. The first approach is to produce ? whenever a type error is encountered. This technique benefits from simplicity, but violates the static gradual guarantee, since adding type information can trigger an approximation that prevents a run-time type error. To avoid this issue, an alternate approach is presented where dynamic type errors are represented explicitly as terms in gradual programs.

1.3.2.4 Gradual Typing Aects Equality

To ensure type safety, gradual programs must perform run-time checks in programs with imprecise types. For non-dependent gradual typing, these checks involve comparing types at run time. However, with dependent types, such a comparison needs to compare type indices, since these may occur within types. In dependent types, any term can serve as a type index, including functions, proofs, and other higher-order structures for whom equality is not decidable.

For static type checking, dependently typed languages typically use different notions of equality in different places: definition equality compares terms syntactically at compile time, while propositional equality denotes the first-class equalities that can be produced or consumed by dependently typed functions. These notions may or may not agree with observational equivalence, whether two terms produce the same result in all contexts.

The reflection of compile-time equality into run-time checks means that, to design a gradual dependently typed language, one must specify precisely how the different notions of equality interact. This dissertation explores two different points in this design space, giving one language where run-time checks use syn-
tactic equality, and another that uses observational equivalence. The former is much simpler, but has the unfortunate property that observationally-equivalent static terms may behave differently when used in the gradual language. The latter is more complex, but preserves static equivalences, and supports a rich notion of gradual equality proofs that can be checked either statically or at run time.

1.3.2.5 Compilation Is Non-Trivial

For a gradual the language to be usable in practice, it should be possible to compile it to (relatively) efficient machine code, and to type check it relatively quickly. Both of these goals are complicated by gradual dependent types, since a term’s actual type might not be known until run time, and type checking involves normalizing and comparing arbitrary terms. Since writing an efficient dependently typed compiler is a monumental task, our solution is to translate gradual dependent types into an existing static dependently typed language. In addition to piggybacking off of another language’s compiler and type checker, such a translation establishes a model of the gradual language and enables mechanization of its metatheory in the static target language. This model provides a path toward a denotational semantics: if the core theory of the target language has a denotational semantics, then a denotational semantics for the gradual language can be obtained via the translation.

1.4 Thesis

Our thesis is that the design of a gradual dependently typed language is feasible and useful\(^2\). In this dissertation, we address the challenges and goals of Section 1.3.2, developing a fully gradual language for dependently typed programming as follows:

- Chapter 2 discusses the background knowledge and related work to support the subsequent chapters, discussing gradual types (Section 2.1), dependent types (Section 2.2), and existing efforts to integrate dynamic or gradual ideas into dependently typed languages (Section 2.3);

\(^2\)This formulation of a thesis statement is borrowed from [99]
• **Chapter 3** gives a foundational calculus for gradual dependent types, introducing the unknown term \( ? \) along with its static and dynamic semantics. We address the problem of decidable type checking using a novel technique called *approximate normalization*, where separate semantics are used for compile-time and run-time normalization of terms. Run-time checks based on definitional equality ensure the safe execution of programs.

• **Chapter 4** reviews the the Gradual Calculus of Inductive Constructions (GCIC), a gradual cast calculus originally presented by Lennon-Bertrand et al. [87]. Though the contents of this chapter are not a contribution of this dissertation, it contains the calculus upon which later chapters are built. GCIC uses features an unknown term with similar semantics to the calculus from Chapter 3, but additionally supports inductive data types. The GCIC authors show that no gradual language can simultaneously embed the static the Calculus of Inductive Constructions (CIC), have strong normalization, and a strengthening of the gradual guarantees. Accordingly, GCIC comes in three variants, each sacrificing one of the above properties while fulfilling the other two.

• **Chapter 5** extends the GCIC variant lacking strong normalization with a gradual notion of propositional equality. That is, we present a type containing gradual proofs that two given terms are equal. These equality proofs dynamically track consistency information about the equated terms, so that when a program violates the specification given by its types, a run-time error can be flagged. This chapter departs from the approach of Chapter 3, ensuring that run-time checks preserve static observational equivalences by deferring checking of functions, instead of using syntactic comparison. Because it is based on a GCIC variant without strong normalization, this chapter does not address the issue of decidable type checking.

• **Chapter 6** presents a syntactic model of a gradual language, that is, a semantics-respecting translation into a static dependent language [17]. It combines the work of Chapters 3 and 5 to give a language that has decidable type checking, inductive types, and gradual propositional equality. Though
we cannot get around the impossibility result of Chapter 4, we show a way to regain decidable type checking without needing the restrictions used by the various \textsc{gcic} variants.

The syntactic model serves three purposes.

1. The model lets us mechanize the metatheory of our gradual calculus, since proofs can be written in the static target. To capture exact runtime semantics, we use Guarded Type Theory (\textsc{gtt}) \cite{14, 15, 140} to soundly model unbounded recursion and potentially non-terminating terms.

2. The model lets us prove strong normalization for approximate normalization, by showing that the static target, in which all terms terminate, can simulate approximate normalization of any term. We overcome several limitations from Chapter 3. In particular, we can prove normalization even for code involving inductive types and large elimination.

3. The gradual-to-static translation serves as an implementation strategy. Gradual terms can be compiled by translating to an existing static dependently typed language, then using its compiler to generate machine code. Likewise, the conversion check for gradual terms can first translate those terms to the static target and check if the results are convertible.

- \textbf{Chapter 7} concludes the dissertation, describing limitations of the research, along with possible directions for future work.
Chapter 2

Background and Related Work

The research in this dissertation involves combining gradual types and dependent types, both of which have a large body of work behind them. As such, we build on the work that has developed these two concepts separately, as well as the much smaller body of work that has gone into introducing type dynamism into dependently typed languages.

2.1 Gradual Typing

Though a variety of systems for optional and flexible type systems have been developed, this thesis focuses on a notion of gradual typing based around an unknown type \( ? \) and a notion of type imprecision.

2.1.1 Imprecise Types

Dynamically typed parts of programs can be represented through imprecise types. Specifically, gradual languages feature a type \( ? \) called the unknown type. A term with type \( ? \) can be used in any context, so dynamically typed computations can be embedded into gradual programs using \( ? \) in type ascriptions. Because a term of type \( ? \) can be used at any type, \( ? \) can express different levels of imprecision. Terms of type \( ? \rightarrow N \) are functions returning numbers, but can take arguments of any type. Likewise, \( N \rightarrow ? \) is the type for functions that take a number but produce a dynamically typed result.
2.1.1.1 The Gradual Consistency Relation

In a statically typed language, checking whether a program is well typed relies on a notion of type equality. For example, given a function

\[ \text{double} : \mathbb{N} \rightarrow \mathbb{N} \]

the type checker verifies that \( \text{double} \) is well typed by comparing the domain of \( \text{double} \) to the type of \( 2 \). In this case, both are \( \mathbb{N} \), so the call is well typed.

However, equality is too restrictive when checking gradual types. Consider some dynamically typed function \( f \) of type \(? \rightarrow \mathbb{N} \). Then \( f \) should be well typed. But, if we compare the domain type to the argument type, \(? \) and \( \mathbb{N} \) are not equal.

The solution is to replace checks of type equality with a new relation, consistency, written \( T_1 \approx T_2 \). Consistency between types represents whether any static type exists that could plausibly be used in place of both types. The unknown type \(?\) is consistent with any type \( T \), since the actual type of the result of a \(?\) typed computation may be \( T \). Composite types are consistent with one another if their parts are. For example, \( S_1 \rightarrow S_2 \approx T_1 \rightarrow T_2 \) if and only if \( S_1 \approx T_1 \) and \( S_2 \approx T_2 \). The equality checks in a static type system correspond directly to the consistency checks of a gradual one. The gradual consistency relation is symmetric and reflexive, but notably, it is not transitive. Since \(?\) is consistent with every type, transitivity would imply that all types were consistent to one another, and no types were statically checked.

2.1.1.2 Gradual Precision and Composition

Consistency is related to precision, a partial ordering \( \sqsubseteq \) on gradual types. For any \( T_1, T \sqsubseteq ? \) holds, and \( T_1 \sqsubseteq T_2 \) holds for composite types \( T_1 \) and \( T_2 \) whenever the parts of \( T_1 \) are each more precise than the corresponding parts of \( T_2 \). For example, \( \text{Nat} \sqsubseteq ? \) and \( \text{Bool} \rightarrow ? \sqsubseteq ? \rightarrow ? \). Unlike a subtyping relation, precision is not contravariant on the left of a function arrow: \( S_1 \rightarrow S_2 \sqsubseteq T_1 \rightarrow T_2 \) holds if and only if \( S_1 \sqsubseteq T_1 \) and \( S_2 \subseteq T_2 \).

A related concept is the precision meet of two types, which provides a way of composing the information of two imprecise types: \( T_1 \cap T_2 \) is the least precise type

\[ \text{while} \approx \text{is typically used to refer to isomorphism, in this dissertation we follow Garcia et al. \[67\] and use} \approx \text{to refer to consistency, following the intuition that it is the gradual lifting of equality.} \]
that is as precise as both $T_1$ and $T_2$, if such a type exists. For example, $\text{Bool} \sqcap ?$ is $\text{Bool}$, and $\text{Nat} \rightarrow ? \sqcap ? \rightarrow \text{Bool}$ is $\text{Nat} \rightarrow \text{Bool}$. Typically, the composition $T_1 \sqcap T_2$ is defined if and only if $T_1 \cong T_2$. The meet is useful for composing type information: a term should check against both $T_1$ and $T_2$ if and only if it checks against $T_1 \sqcap T_2$.

The precision relation on types gives a way to compare programs for precision. Term precision, $t_1 \sqsubseteq t_2$, is defined using the precision of their type annotations (ascriptions on function arguments, or general term ascriptions, which we write as $::$). So $(\lambda x : \text{Int} . x) \sqsubseteq (\lambda x : ? . x)$ holds. Term precision lets us reason about changes in program behaviour in terms of changes in program precision, which is key for defining the gradual guarantees in Section 2.1.2.2.

The composition of two types is sometimes written as $T_1 & T_2$. We use the ampersand to refer to composition as an operation, and reserve $\sqcap$ for semantic composition in the model in Chapter 6.

### 2.1.2 Gradual Metatheory

Adding $?$ to a language is not enough to make it a good gradual language. Instead, we want our language to fulfill certain metatheoretic properties, to ensure that it behaves safely and predictably, and that changes in precision do not cause unexpected behaviour.

#### 2.1.2.1 Gradual Type Safety

Consistency lets us type check programs with imprecision, but this necessarily means assigning types to some programs where a term’s type assignment is incompatible with its context. Hence, type checking is only half of the gradual typing story. For a gradual language to be safe, dynamic checks must ensure that no type-unsafe operations are performed [136]. This safety is a key concept of gradual typing: type checks are always performed, but they may be performed at compile time or at run time, depending on how much type information is statically available.

Many presentations of gradual typing ensure run-time safety using a *cast calculus*. Under the cast calculus approach, gradual programs are elaborated into a language with type annotations and explicit casts. The semantics of a cast calcu-
lus specify the result of casting between any pair of types, and whether that result is a run-time type error. We use a cast calculus to define the semantics of gradual dependent types from Chapter 4 onward.

An alternate approach to ensuring safety is the evidence-based approach of the Abstracting Gradual Typing (AGT) [67] framework, which we explain in Section 2.1.3.3.

2.1.2.2 The Gradual Guarantees

Many languages and systems claim to have some form of optional or mixed static-dynamic typing, but they vary widely in how safe they are and how smoothly terms can be moved from dynamic to static or vice-versa. Greenman and Felleisen [70] describe a generalized spectrum of type soundness, ranging from fully safe gradual types to erasable types with no run-time checks. Siek et al. [136] identify refined criteria for what makes a language, in their eyes, truly gradual. We follow these latter criteria.

In addition to gradual type safety (which they call soundness), the key property of a gradual language is:

- If a term \( t : T \), and \( t \subseteq t' \), then \( t' : T' \) for some \( T' \) where \( T \subseteq T' \).

- If a term \( t : T \) runs successfully with result \( v \), and \( t \subseteq t' \), then \( t' \) runs successfully with result \( v' \) where \( v \subseteq v' \). If \( t'' \subseteq t \) where \( t'' : T'' \) and \( T'' \subseteq T \), then either \( t'' \) runs to some \( v'' \) where \( v'' \subseteq v \), or \( t'' \) raises a dynamic type error.

These guarantees, referred to as the static gradual guarantee and dynamic gradual guarantee respectively, have critical implications for the process of incrementally increasing the precision of annotations in a program. The static guarantee is useful because of the contrapositive: if a program does not type check, then it is because some types are fundamentally incompatible, not because an ascription is missing. Likewise, the dynamic guarantee captures the property that, modulo precision, type ascriptions do not change the behaviour of a program. Static or dynamic type errors are always caused by incompatible types, never by imprecise type information. The static guarantee goes in one-direction, but the
dynamic guarantee is bidirectional: since increasing the precision of type information may result in an ill typed program, the static guarantee does not address this case.

An interesting property is that, in a language satisfying the dynamic gradual guarantee, run-time type errors must not be caught. Though this may seem undesirable, the ability to catch dynamic type errors could mean that changing a type annotation of a program could result in completely different behaviour. With the gradual guarantee, a programmer can add or remove type signatures with the knowledge that the program’s behaviour does not change (modulo uncatchable type errors), so migrating from dynamic to static types is easy.

### 2.1.2.3 Embedding-Projection Pairs

Related to the gradual guarantees is the embedding-projection pairs (EP-pairs) property. The EP-pairs property relates type precision to casts. If $T_1 \subseteq T_2$, then the casts $\langle T_2 \leftarrow T_1 \rangle$ and $\langle T_1 \leftarrow T_2 \rangle$ form an EP-pair if:

- Casting $t$ from $T_1$ to $T_2$, then casting that back to $T_1$, yields a term that is observationally equivalent to $t$. That is, for all contexts $C : T_1 \rightarrow \text{Bool}$, $C[t]$ evaluates to a boolean equal to $C[\langle T_1 \leftarrow T_2 \rangle (T_1 \leftarrow T_2) t]$;

- Casting $t'$ from $T_2$ to $T_1$, then casting that back to $T_2$, yields a term $t''$ such that, for every context $C : T_2 \rightarrow \text{Bool}$, either $C[t'']$ raises an error, or $C[t']$ and $C[t']$ evaluate to equal booleans.

That is, reducing then increasing precision produces the original term, and increasing then reducing precision produces a term that behaves like the original, except that it might be an error, or raise an error in strictly more contexts. For non-dependent gradual types, the intuition is that reducing precision lets a term be used at a less precise type, but that the result of that cast is in some sense the “same” term as the original. Likewise, increasing the precision of a term should not change its behaviour, but it might increase the precision of run-time type information, and hence "remember" what type it was cast to, causing future operations to trigger an error.
New and Ahmed [105] define *graduality* as the semantic version of the gradual guarantee: two precision-related terms produce precision-related results in any context. The existence of EP-pairs between any two precision-related types implies graduality. Just as parametricity strengthens type safety, and proves it as a corollary, graduality strengthens the gradual guarantees and, as the authors show using a logical-relations argument, graduality proves the guarantees as a corollary.

For gradual dependent types, EP-pairs are even more important than with simply-typed languages: since ? is a term as well as a type, EP-pairs guarantee that casts don’t needlessly produce ?. However, Lennon-Bertrand et al. [87] showed that no conservative extension of the Calculus of Inductive Constructions can have both EP-pairs and strong normalization. In Chapter 7, we describe a weaker version of EP-pairs that captures the idea that run-time casts do not needlessly lose information in gradual dependently typed languages, and conjecture that they can be adapted to the model we give in Chapter 6.

2.1.2.4 Gradual Full Abstraction

Jacobs et al. [81] describe another property that is important for gradual languages: a fully abstract embedding between a static language and its gradual counterpart. Full abstraction means that two static terms are observationally-equivalent if and only if the embeddings of those terms are equivalent in the gradual language. Full abstraction enables the programmer to fearlessly refactor static code: since terms that are statically equivalent are also gradually equivalent, they know that if a refactoring is safe statically, it is also safe in the gradual language.

Full abstraction has a subtle interaction with gradual dependent types: because types may be indexed by functions, any two functions that are extensionally equal (agree on all inputs) must be observationally equivalent in the gradual language. Specifically, two extensionally equal functions must raise type errors in exactly the same contexts, a goal which is incompatible with reflecting checks of definitional equality to run time. We discuss an approach to preserving equivalences in Chapter 5, and a potential approach to prove a weaker version of full

---

2Lennon-Bertrand et al. [87] refer to the existence of EP-pairs as graduality, since for dependently typed languages it is a stronger statement than the gradual guarantee.
abstraction in Chapter 7.

2.1.3 Abstracting Gradual Typing

Thus far, we have not presented a general way to make a static language gradual. Ad hoc notions of gradual typing enabled the integration of gradual types and many programming language features, including mutable state [137], union and intersection types [24, 145], ownership types [128], effects [9], parametric polymorphism [4, 147], security types [64, 146], and session types [80]. However, without a unified technique for converting a static type system into a gradual one, proofs of safety and the gradual guarantees are lengthy and ad hoc. Garcia et al. [67] seeks to remedy this by introducing the framework of Abstracting Gradual Typing (AGT).

We use AGT to devise our foundational gradual dependently typed language in Chapter 3.

2.1.3.1 Concretization: The Meaning of Gradual Types

The AGT framework is based around the idea that the meaning of a gradual type should be specified in terms of the set of static types that it could plausibly represent. Consider a statically typed language whose types come from the set \( \text{SType} \). We want to extend this language into a gradual language, whose types come from \( \text{GType} \supseteq \text{SType} \). To specify the meaning of the new gradual types, we define a function \( \gamma : \text{GType} \rightarrow \mathcal{P}(\text{SType}) \). This function is known as the concretization function. Intuitively, \( \gamma(T) \) represents the set of static types that a value of type \( T \) could represent.

Concretization gives a systematic way of deriving relations in gradual typing. Before AGT, consistency and precision were defined ad-hoc, with a number of cases for each type introduction form. With AGT, we define the concretization once by cases, and then say that \( t_1 \equiv t_2 \) if and only if \( \gamma(T_1) \cap \gamma(T_2) \neq \emptyset \). Likewise, \( T_1 \subseteq T_2 \) can be defined as \( \gamma(T_1) \subseteq \gamma(T_2) \).

Once gradual relations have been defined from a concretization function, many static typing rules can be systematically turned into gradual ones. When rules contain repeated metavariables, this repetition induces implicit equality constraints.
on the sub-terms of the judgment. These so-called “pattern matches” can be eliminated by instead using distinct metavariables in each premise, and introducing explicit equality side-conditions these metavariables corresponding to the repetitions in the original rule. To convert such rules from static to gradual, the equality side-condition is replaced with a consistency check. In general, a binary predicate $P$ can be converted to a gradual form $\tilde{P}$ by taking:

$$\tilde{P}(t_1, t_2) \iff \exists S_1 \in \gamma(t_1), S_2 \in \gamma(T_2). P(S_1, S_2)$$

This easily generalizes to $n$-ary predicates.

To see the transformation in action, consider a typing rule for addition:

$$\Gamma \vdash t_1 : \mathbb{N} \quad \Gamma \vdash t_2 : \mathbb{N} \quad \Gamma \vdash t_1 + t_2 : \mathbb{N}$$

This rule is equivalent to the following rule:

$$\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad T_1 = \text{Nat} \quad T_2 = \text{Nat} \quad \Gamma \vdash t_1 + t_2 : \text{Nat}$$

To convert this to a gradual rule, we replace the equality checks with consistency checks:

$$\Gamma \vdash t_1 : t_1 \quad \Gamma \vdash t_2 : t_2 \quad t_1 \equiv \text{Nat} \quad t_2 \equiv \text{Nat} \quad \Gamma \vdash t_1 + t_2 : \text{Nat}$$

This change allows addition for values of type $\rho$ or type $\text{Nat}$ to be well typed.

### 2.1.3.2 Abstraction and Partial Functions

Concretization lets us deal with equalities and other static predicates, but it is not sufficient to convert rules that require types to have a certain shape. Consider the typing rule for function application:

$$\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1 \quad \Gamma \vdash t_1 \cdot t_2 : T_2$$

To avoid repeated metavariables, the rule could require that $t_1 : T'$, and have $T' = T_1 \rightarrow T_2$ as a side-condition. However, when we relax equality to consistency for
the gradual language, the rule is no longer syntax directed, since any number of arrow types may be consistent with the type of $t_1$.

To derive gradual versions of static rules like these, the key is to avoid pattern matching by defining partial functions for decomposing a type. For example, we define a partial domain function as follows:

\[
\text{dom}(T_1 \rightarrow T_2) = T_1 \quad \text{undefined otherwise}
\]

The codomain function cod can be defined similarly. Then, the static application rule can be rephrased as follows:

\[
\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad \text{dom}(T_1) = T_2}{\Gamma \vdash t_1 t_2 : \text{cod}(T_1)}
\]

The gradual language can use a partial function $\tilde{\text{dom}}$ where $\tilde{\text{dom}} (T_1 \rightarrow T_2) = T_1$ and $\tilde{\text{dom}} ? = ?$, but is undefined otherwise. The codomain $\tilde{\text{cod}}$ can be defined similarly. Given these functions, we replace $\text{dom}$ and $\text{cod}$ with their gradual counterparts to obtain a gradual rule for application.

To derive these lifted partial functions in general, we need an abstraction function $\alpha : \mathcal{P} \text{(SType)} \rightarrow \text{GType}$, which serves as the converse of concretization, taking sets of static types to a single representative gradual type. The function $\alpha$ should be sound, where $X \subseteq \gamma(\alpha(X))$ for any non-empty $X \in \mathcal{P} \text{(SType)}$, as well as optimal: $\alpha(X) \subseteq t$ whenever $X \subseteq \gamma(t)$ for non-empty $X$. These conditions ensure that the abstraction returns a type capturing all possibilities of the input set, but that it respects the subset ordering of the concretization sets.

The abstraction $\alpha$ gives a way to lift arbitrary partial functions from static to gradual. If $F : \text{SType} \rightarrow \text{SType}$, then we define its lifting $\tilde{F} : \text{GType} \rightarrow \text{GType}$ as $\tilde{F}(t) = \alpha(F(\gamma(t)))$: that is, we apply $F$ element-wise on each element of $\gamma(t)$, then abstract back to a single gradual type. This can be generalized to functions with any number of arguments.
2.1.3.3 Evidence for Dynamic Semantics

Though concretization and abstraction give us the tools to systematically derive the typing rules for a gradual language from its underlying static language, we would also like to execute gradual programs, and ensure that they either succeed or fail with a run-time type error. We do so with an intermediate language where run-time terms may be ascribed evidence, written using blue, bold serif font. For two types $T_1, T_2$, the consistency judgment $T_1 \equiv T_2$ holds if and only if there exists some type in the overlap of their concretizations. If $\alpha$ is sound and optimal, then there exists some $T_3 \subseteq T_1 \cap T_2$. We call such a witness evidence of their consistency, and write $\langle T_3 \rangle \vdash T_1 \equiv T_2$. (The angle brackets $\langle \rangle$ denote that we are treating their contents not as a type, but as information that is carried around at run-time).

The key feature of gradual consistency, and the reason that run-time type errors may still happen with gradual types, is that consistency is not transitive. We may accept a boolean where a number is expected, provided we first ascribe that boolean the type $\top$, which is consistent with both Int and Bool. However, $\text{Int} \neq \text{Bool}$. AGT generates a gradual type system by replacing static equality with consistency. But the transitivity of equality is a key property needed to establish the safety of a static language. AGT accounts for this using consistent transitivity: if $\langle S_1 \rangle \vdash T_1 \equiv T_2$ and $\langle S_2 \rangle \vdash T_2 \equiv T_3$, then $\langle S_1 \cap S_2 \rangle \vdash T_1 \equiv T_3$, if $S_1 \cap S_2$ is defined. That is, given evidence that $T_1 \equiv T_2$ and $T_2 \equiv T_3$, consistent transitivity informs us how we can compose that evidence to determine whether $T_1 \equiv T_3$. If the meet is undefined, then we cannot conclude that the transitive composition of the two consistency relations holds, given the available evidence.

Evidence is useful for preserving the well-typedness of terms under reduction. Where surface syntax allows $t : T_2$ whenever $t : T_1$ and $T_1 \equiv T_2$, the evidence language instead only allows this for terms of the form $e : T$, where $e \in (T_1 \cap T_2)$. That is, all applications of consistency in the evidence language are explicitly represented in the syntax, and are relevant to reductions. In places that the static proof of type preservation uses the transitivity of equality, the gradual proof uses con-

---

3 Here we describe evidence for consistency, the gradual lifting of equality. If a language uses a more complex comparison relation in place of equality, such as subtyping, the approach can be extended, as described by Garcia et al. [67].
sistent transitivity, ensuring that either typing is preserved or the reduction fails. At any point, the evidence ascription on a term represents the most precise information we know about its type, and this information increases monotonically. This approach significantly simplifies the proof of gradual type safety: evidence is composed specifically to preserve typing, making subject reduction easy to show. Because consistent transitivity is a monotone operation, the gradual guarantee can be shown to hold using a straightforward simulation argument.

2.2 Dependent Types

In this section, we provide relevant background knowledge on dependently typed languages. Specifically, we focus on full-spectrum dependent types, in the style of the Calculus of Inductive Constructions (cic) and Martin-Löf Type Theory (mltt), where types and terms inhabit a single syntactic category: types may contain expressions, and expressions can produce types as a result. Systems using this style of dependent types include Coq [12], Agda [109], Lean [46], Dependent Haskell [55], Cur [28] and Idris [19]. Types in these languages differ from the lightweight dependent types of Dependent ML [157] and its successor ATS [156], or the refinement types of Liquid Haskell [150], where the interaction between types and terms is restricted. F★ [142] mixes the two styles, using SMT to discharge proof obligations when possible, but falling back to full dependent types with manual proofs when a solution cannot be automatically found.

Though cic [116] and mltt [93, 94] are widely known formalizations of dependent types, we avoid committing to either, since ideas from both are useful to us. Various presentations of cic and mltt disagree on precisely what distinguishes them: for example, Lennon-Bertrand [86] present a bidirectional version of cic without indexed types, and homotopy type theory uses a variant of mltt with cumulative universes [149], both of which differ from the “standard” presentations. For our purposes, the main distinction between the two is that cic uses

---

4Some variations of type theory use a Universe à la Tarski, where types and terms inhabit separate categories, but a class of terms known as codes reflects types to the term level. We define our own Universe à la Tarski in Chapter 6, but assume that our underlying type theories have a Universe à la Russell where types and terms are unified.
an untyped rewrite relation to define when two terms are convertible, whereas MLTT uses a typed judgmental equality to compare terms for convertibility\(^5\).

### 2.2.1 Essential Dependent Types

We begin by introducing dependent functions and pairs, which are the key type formers that enable value dependency.

#### 2.2.1.1 The Dependent Function Space

The central feature of a dependently typed language is the *dependent function type*. The simple function type \(T_1 \to T_2\) denotes the type of functions that take an argument of type \(T_1\) and return a value of type \(T_2\). The dependent function type is written as \((x : T_1) \to T_2\), where critically, \(x\) may be free in \(T_2\). The meaning of the dependent function type is perhaps best seen with (a version of) the typing rules for functions and applications.

\[
\begin{align*}
\Gamma, x : T_1 \vdash t : T_2 & \quad \Gamma \vdash \lambda x. t : (x : T_1) \to T_2 \\
\Gamma \vdash t_0 \vdash t_1 : T_1 & \quad \Gamma \vdash t_1 : T_1 \\
& \quad \Gamma \vdash [t_1/x] T_2
\end{align*}
\]

A function is well typed if its body has the stated return type when the parameter is added to the typing environment (which is why the parameter may occur in codomain). If we apply a function to an argument, the return *type* of that application may depend on the *value* of the argument, thus the name “dependent function.”

The dependent function type is sometimes written as \(\Pi(x : T_1). T_2\), though we prefer the arrow notation to reflect the textual notation of programming languages, particularly Agda and Idris. It is sometimes called a dependent product type, because it is the generalization of product types to infinite index sets, though we prefer “function type” to avoid confusion with dependent pairs. The simple product type \(T_1 \times T_2\) is equivalent to \((x : \mathbb{B}) \to (\text{if } x \text{ T}_1 \text{ T}_2)\), where \(\mathbb{B}\) is the type of booleans.

\(^5\)Arguably the greatest distinction between CIC and MLTT is an impredicative sort, but Lennon-Bertrand et al. [87] call GCIC a variant of CIC without an impredicative sort, so we avoid this distinction.
2.2.1.2 Dependent Pair Types

The dual of the dependent function type is the dependent pair type, which we write $(x : T_1) \times T_2$, where $x$ may be free in $T_2$. Just as the dependent function type lets the return type depend on the value given for the argument, dependent pairs allow the type of the second element to depend on the value of the first. We can see this in the typing rules for dependent pair accessors:

$$
\Gamma \vdash t : (x : T_1) \times T_2 \\
\Gamma \vdash \pi_1 t : T_1 \\
\Gamma \vdash \pi_2 t : \pi_1 t / x : T_2
$$

The dependent pair type is sometimes written as $\Sigma(x : T_1). T_2$, and is sometimes called the dependent sum type, since it generalizes sum types to an infinite index set. We prefer the term “pair”, since dependent pairs can model both simple sums and simple products: $(x : T_1) \times T_2$ where $x$ is not free in $T_2$ gives the simple product type, and $(x : B) \times (\text{if } x T_1 T_2)$ gives the simple sum type. This dual nature of the dependent pair as both product and sum makes it ideally suited for generically representing inductive data types [29]: though inductive types are traditionally sums-of-products, dependent languages can represent them as sigmas-of-sigmas.

Several features of dependent types can be desugared into dependent pairs. Record types, with named fields whose types can depend on the values of previous fields, can typically be desugared into nested dependent pairs. We use records extensively in Chapter 6. Similarly, simple sums can be expressed as dependent pair. We write $T_1 + T_2$ for the non-dependent sum type given by $(x : B) \times \text{if } x T_1 T_2$, with injections $\text{inj}_1$ and $\text{inj}_2$ and a non-dependent eliminator $\text{sumCase}$.

2.2.1.3 Type-Term Overlap

What does it mean for a value to occur in a type? For functions, variables denoting the value of a function’s argument can occur free in the codomain of a function type. In a full-spectrum language, types and terms inhabit a single syntactic category, or there is a special type of codes that reflects types to the term level. So using function applications or other expressions within a type is permitted syntactically. Since types are terms, they are also assigned types: if $t : T$ then
Likewise, expressions may take types as arguments, and produce types as a result. For instance, \( \text{Nat} \rightarrow \text{Type} \) is a valid type. In Section 1.3.1.1, we saw that \( \text{Vec} \) had a type of \( \text{Type} \rightarrow \mathbb{N} \rightarrow \text{Type} \): \( \text{Vec} \) takes a type and a number, and produces another type as a result. In general, if \( F : \text{Nat} \rightarrow \text{Type} \), then \( (x : \text{Nat}) \rightarrow F \; x \) is a well-formed type. We say that \( x \) is the index of \( F \) in this case. Such functions may be formed using \( \lambda \)-abstraction, or they may be declared with inductive types or families (Section 2.2.2).

2.2.1.4 The Curry-Howard Correspondence

The true power of dependent types lies not only in the ability to write programs with them, but also the ability to write proofs about those programs. The Curry-Howard correspondence [40, 76] is a view of type theory where types correspond to propositions, and terms correspond to proofs. Inhabited types correspond to theorems. Function, product and sum types correspond to implication, conjunction and disjunction respectively.

The dependent product and pair types fit nicely into the Curry-Howard interpretation, acting as quantifiers. The type \( (x : T_1) \rightarrow T_2 \) corresponds to the proposition \( \forall (x \in T_1). T_2 \): given a function producing a proof of \( T_2 \) for each possible argument \( x \) in \( T_1 \), we can conclude the proposition holds universally. Similarly, \( (x : T_1) \times T_2 \) corresponds to \( \exists x \in T_1. T_2 \). That is, if a pair containing \( t : T_1 \) and a proof that \( T_2 \) holds for \( t \), then \( t \) serves as the existential witness for the proposition \( \exists x. T_2 \). Consider \( (3, \text{refl}) : (x : \text{Nat}) \times (x + 1 \equiv 4) \). Here 3 serves as a witness that there exists a number one less than 4, and \( \text{refl} \) is the reflexive proof that \( 3 + 1 \equiv 4 \).

Dependent types use types to precisely specify function behaviour. Preconditions of functions can be specified by adding arguments that specify certain predicates on the inputs that must hold. Likewise, postconditions can be specified by returning dependent pairs containing values along with proofs that those values satisfy the desired property. Conversely, dependent types allow for flexibility when defining functions: different inputs may produce outputs of different types, as long as some expression can describe the output type in terms of the input value.

With dependent types, the programmer can work within the rigidity of precise
specifications, constructing proofs programmatically to convince the type checker
the preconditions of a given call hold. Thus, dependent types provide a sound and
complete form of verification: if there exists a proof that a property holds, then
our types can use that fact, so long as the programmer can construct such a proof.
This manual construction is in contrast to automated verification techniques, like
symbolic execution or model checking, where the user need not provide proofs,
but as a result all analyses produce false positives or false negatives. The cost of
this power, however, is that the typing discipline is rigid, and that programmers
must construct proof terms whenever a function requires them.

2.2.1.5 Universes and Predicativity

Above, we saw the type \texttt{Type}, which is the type of types. What should the type of
\texttt{Type} be? Allowing \texttt{Type : Type} enables Girard’s Paradox [37], the type-theoretic
analog to Russell’s paradox that provides a closed inhabitant of any type. Compu-
tationally, some proofs in the language corresponds to non-terminating compu-
tations, which causes two problems. First, because values may occur in our types,
we must compare terms during type checking, usually by conversion to some
normal form. Having non-termination makes type checking undecidable, since
determining equivalence of terms can be used to determine whether a given Tur-
ing Machine halts. Second, proofs under the Curry-Howard correspondence are
no longer meaningful: the corresponding logic would be inconsistent since we
could write a (possibly non-terminating) proof for every proposition.

The standard way to avoid Girard’s Paradox is to introduce a \textit{universe hierar-
chy}. For a term \(t : T\), where \(t\) is not a type (or something as “large” as a type),
\(T : \texttt{Type}_0\). Then \(\texttt{Type}_0 : \texttt{Type}_1, \texttt{Type}_1 : \texttt{Type}_2, \text{etc.}\) Throughout, we write \texttt{Type}
to mean \(\texttt{Type}_0\).

We focus on languages in which quantification is \textit{predicative}: if \(T_1, T_2 : \texttt{Type}_\ell\),
then \((x : T_1) \rightarrow T_2 : \texttt{Type}_\ell\) and \((x : T_1) \times T_2 : \texttt{Type}_\ell\). So \(T_1\) and \(T_2\) may only refer to
\(\texttt{Type}_{\ell'}\) for \(\ell' < \ell\). Impredicative dependently typed systems do exist, such as the
Calculus of Constructions [39]. Such systems still disallow \(\texttt{Type}_\ell : \texttt{Type}_{\ell'}\), but let
\((X : \texttt{Type}_\ell) \rightarrow T_2\) have type \(\texttt{Type}_\ell\). However, great care must be taken to ensure
that the formulation of inductive types or the use of postulated axioms does not
introduce inconsistency due to impredicativity.

A related concept to universe levels is *cumulativity*. We say a language is cumulative where, whenever \( t : \text{Type}_\ell \), we also have \( t : \text{Type}_{\ell+\ell_0} \) for any \( \ell, \ell_0 \).

### 2.2.2 Inductive Families and Pattern Matching

Though dependent functions and pairs give us universal and existential quantification, we lack a key tool from logic: induction. Here, we describe how users of dependently typed languages can define their own types, and how they can prove facts about them using induction.

#### 2.2.2.1 Inductive Types

Many dependently typed languages allow for the definition of *inductive types*, which combine aspects of recursive types from simple type theory [22], sum types, and dependent pairs. Like algebraic data types in Haskell or ML, inductive types are declared by specifying a name for the type, along with zero or more constructors. For example, the type of natural numbers can be declared as follows, using Agda style notation:

```agda
data Nat : Type where
  Z : Nat
  S : Nat → Nat
```

Here, `Nat` is a type, with two data constructors, `Z` and `S`. Data constructors may take any arguments, but must return the declared type. Self reference is allowed: `Nat` can occur as a type for `S`. However, such self reference must be *strictly positive* (never in the domain of a function-typed field) to maintain logical consistency. Without strict positivity, one can define `T` that is isomorphic to `T → T`, embedding the untyped lambda calculus and enabling non-termination.

An inductive type may have one or more *type parameters* over which it is
generic. For example, the classic list type can be declared as follows:

```haskell
data List (X : Type) : Type where
  Nil : List X
  Cons : X → List X → List X
```

Here, `List` is a type constructor with type `Type → Type`. The parameter is declared and named at the start of the declaration, and the parameter is used in the return type of each constructor.

Inductive types and algebraic data types differ in two main ways. First, constructors are assigned dependent function types, enabling existential constraints to be encoded in constructor types. For example, the dependent pair type from Section 2.2.1.2 may be rephrased as an equivalent inductive type:

```haskell
data Ex (X : Type) (P : X → Type) : Type where
  Pack : (x : X) → P x → Ex X P
```

The second difference is that, unlike in Haskell or ML, inductive types cannot be consumed using unrestricted pattern matching or general recursion. To ensure logical consistency, inductive types must either be consumed using eliminators, which correspond to induction principles from logic, or using dependent pattern matching and structural recursion. We explain both of these in Section 2.2.2.4.

### 2.2.2.2 Indexed Inductive Families

Indexed inductive families, commonly called inductive families, further extend the power of inductive types. Parameters for inductive types must occur uniformly in the return types of all constructors. Inductive families relax this restriction by letting the type constructor additionally take a different kind of type argument: indices. For inductive families, indices are like parameters, but each data constructor may specify values for the indices of its return type. We call these type families because one definition declares an entire family of types, one for each different combination of indices. Unlike a parameterized type, the value of the index constrains which constructors could have been used to construct a value.
from a type in the family. 
Recall the length-indexed vectors of Section 1.3.1.1:

data Vec (X : Type) : (n : Nat) → Type where
  Nil : Vec 0 X
  Cons : (m : Nat) → X → Vec X m → Vec X (1 + m)

The type constructor Vec has type Type → Nat → Type. Here, X is a parameter, but n is an index, which we show using Agda’s convention of keeping parameters to the left of the colon. Nil only produces vectors of length zero, and Cons only produces vectors with length one more than their tail. So Vec T n is a distinct type for each n, which is only ever inhabited by vectors with exactly n elements. Type indices let us reason about the concrete values of a vector. Given a value of type Vec T (n + 1), we know that it cannot possibly be Nil if it is well typed, since the Nil constructor does not return a value of this type.

2.2.2.3 Propositional Equality

Where inductive types give the Curry-Howard equivalent of inductively defined sets, inductive families provide a version of the inductive relations commonly used to define judgments in the theory of programming languages. Constructors may use their arguments in multiple indices, allowing for the same kinds of constraints as inference rules.

A key relation is propositional equality, since, as we explain in Section 2.2.2.6, all other inductive families can be built from inductive types and propositional equality. The type for propositional equality can be expressed as an inductive family:

data Eq (X : Type) : X → X → Type where
  refl : (y : X) → Eq X y y

We usually write the type Eq T t₁ t₂ as t₁ =_T t₂, omitting T when it is clear from context. For any two values of type T, the type t₁ =_T t₂ is well-formed, expressing the proposition that t₁ and t₂ are equal. Since refl is the only constructor for this
type, the type is only inhabited when \( t_1 \) and \( t_2 \) are actually equal. Specifically, \( t_1 \) and \( t_2 \) must have syntactically equal normal forms, modulo other equality proofs that are in scope.

Propositional equality is very useful when programming with dependent types, because we can express equality not only between concrete values, but also between terms containing variables. Dependent functions that can take arguments whose types are equalities, used to express the precondition that an argument is equal to some value. Likewise, an equality proof can be provided alongside the result of computation, giving a certificate that a certain property holds for the result of a function call.

Chapter 5 is largely concerned with how to integrate propositional equality into a gradually typed language, and how to track and store dynamic consistency information in equality proofs.

### 2.2.2.4 Deconstructing Inductive Types

Under Curry-Howard, deconstructing a value of an inductive type corresponds to a proof by cases. By inspecting which constructor was used to construct a value, we gain more refined information about the type, that can either be used to prove invariants or to derive a contradiction, effectively showing that a branch is unreachable. Likewise, recursion involving inductive types gives the Curry-Howard analogue to induction.

The two main strategies for deconstructing inductively defined data are *eliminators* and *dependent pattern matching*.

**Eliminators**  An eliminator is a function built into a type theory that recursively decomposes a value of a recursive data type. For example, the eliminator for the natural number type from Section 2.2.2.1 has the following type, and satisfies the
following equalities:

\[
\text{natElim} : (P : \text{Nat} \rightarrow \text{Type},) \rightarrow (P \, Z) \rightarrow \\
((n : \text{Nat}) \rightarrow (P \, n) \rightarrow P(S \, n)) \rightarrow (x : \text{Nat}) \rightarrow P \, x
\]

\[
\text{natElim} \, m \, z \, s \, 0 = z
\]

\[
\text{natElim} \, m \, z \, s \, (S \, n) = s \, n \, (\text{natElim} \, m \, z \, s \, n)
\]

The eliminator has a few components:

- It begins by taking a motive \(P\), which is a type family (i.e. function producing a type) that takes the value being eliminated and returns the type of what we are trying to produce. For any scrutinee \(t\) (i.e., the term whose form we are branching on), elimination produces a value of type \(P \, t\): the \(Z\) branch produces \(P \, Z\) and the \(S \, n\) branch produces \(P \, (S \, n)\). Because the motive takes the scrutinee as an argument, the result type of elimination may differ for each potential argument;

- We require a value of our motive applied to the base case \(0\);

- We also require a function that produces a value of our motive type for \(n + 1\), given one for \(n\);

- Given the above arguments, the eliminator takes the scrutinee, and produces a value of the motive at that number.

This eliminator corresponds to the principle of mathematical induction on natural numbers. It also provides a programming construct for both branching and recursion: given a handling function for each constructor of an inductive type, the eliminator checks which constructor the given value uses and proceeds accordingly.

In general, an eliminator for a type gives a fold function over that type, and can be defined for any inductive type, indexed or not. For example, the eliminator for vectors resembles a conventional list fold, except that the return type can depend
on the value of the given vector:

\[
\text{vecElim} : (X : \textbf{Type}) \rightarrow (P : (n : \textbf{Nat}) \rightarrow \textbf{Vec} X n \rightarrow \textbf{Type}) \rightarrow (P (\textbf{Nil} X)) \\
\rightarrow ((n : \textbf{Nat}) \rightarrow (h : X) \rightarrow (t : \textbf{Vec} X n) \rightarrow (P n t) \rightarrow P (\textbf{Suc} n) (\textbf{Cons} X n h t)) \\
\rightarrow (n : \textbf{Nat}) \rightarrow (v : \textbf{Vec} X n) \rightarrow P n v
\]

The result is a function which produces \( P n v \) for any \( n : \textbf{Nat} \) and \( v : \textbf{Vec} X n \). The eliminator for a type family lets us prove not only properties for all values of a type, but for all values of each type in the family.

Dependently typed eliminators are useful because they guarantee termination: recursion is built into the theory, and is always on strictly smaller values. However, they are cumbersome to program with, since motives must be specified explicitly. Dybjer [50] describe the general form of an eliminator for an inductive family.

**Dependent Pattern Matching**  Dependent pattern matching provides the same functionality as eliminators, but with less syntactic overhead. When a function is defined, the programmer may specify multiple clauses with the possible patterns that the arguments may take, akin to the case expressions of Haskell or match in ML.

Unlike in Haskell and ML, inductive families may have type indices, so matching a value to a specific constructor may provide more refined knowledge of the argument type. In the branch for a constructor, the index values of a match’s scrutinee are *unified* with the concrete index values for said constructor, where program variables are treated as unification variables. The substitution produced by this unification is applied to all variables free in the clause’s result, as well as to the type of said result. In cases where no unifier exists, we know that no well typed input to the function can reach that branch, so those cases may be safely elided.

Unlike eliminators, recursion is not built into the structure of pattern matching. To alleviate this, a named function may be allowed to call itself. However, not all pattern matches defined this way are terminating. Because of this, matches are usually fed to an external termination checker [108]. Such a checker ensures
that each recursive function is made on a strictly-smaller argument. The termination check is necessarily conservative: since we cannot solve the halting problem, programs exist that halt but cannot be proved to halt.

Pattern matches are as expressive as eliminators, and can be strictly more expressive depending on the formulation. Goguen et al. [69] give a language with dependent pattern matching that is equivalent to a combination of eliminators and Streicher’s K axiom [141], which asserts that any proof of \( x =_T x \) is refl. While this principle seems obvious from refl being equality’s only constructor, it cannot be derived using the eliminator for equality. Axiom K is equivalent to UIP, which asserts that any two proofs of equality are themselves equal. Recent work has shown how to formulate dependent pattern matching without implying axiom K [34, 35], allowing pattern matching to be used in more theoretical settings.

### 2.2.2.5 Eliminating Equality Proofs

When propositional equality is defined as an inductive type, several useful principles about equalities can be derived from its eliminator. The eliminator itself corresponds to the J-rule from MLTT [94]:

\[
J : (X : \text{Type}) \to (P : (z_1 : X) \to (z_2 : X) \to z_1 =_X z_2 \to \text{Type}) \\
\to ((y : X) \to P y y \text{refl}_X y) \\
\to (z_1 : X) \to (z_2 : X) \to (x : z_1 =_X z_2) \to P z_1 z_2 x
\]

That is, for a motive \( P \) that is parameterized over two values \( z_1, z_2 : X \), and a proof \( \text{pf} : z_1 =_X z_2 \), if we are given a way for any \( y \) to build a motive instance for \( y \) and itself with \text{refl}, then we can produce an instance of the motive for \( z_1, z_2 \) and \( \text{pf} \) whenever \( \text{pf} : x =_T y \).

The \( J \) principle can be used to implement a more familiar substitution principle, which allows any type involving \( y \) to be transformed into one involving \( z \) when \( y = z \):

\[
\text{subst} : (X : \text{Type}) \to (P : X \to \text{Type}) \to (y : X) \to (z : X) \to y =_X z \to P y \to P z
\]
When $X$ is $\text{Type}$ and $P$ is set to the identity, the result is the transport principle$^6$:

$$\text{transport} : (X_1 : \text{Type}) \rightarrow (X_2 : \text{Type}) \rightarrow X_1 =_{\text{Type}} X_2 \rightarrow X_1 \rightarrow X_2$$

That is, we can convert a value of one type to some other type, given a proof that the two types are equal.

Other useful functions that can be derived from $J$ include proofs that propositional equality is symmetric and transitive, as well as a congruence function establishing that applying a function to equal values produces equal results:

$$\text{cong} : (X : \text{Type}) \rightarrow (Y : \text{Type}) \rightarrow (f : X \rightarrow Y) \rightarrow$$

$$(z_1 : X) \rightarrow (z_2 : X) \rightarrow z_1 =_X z_2 \rightarrow f z_1 =_Y f z_2$$

### 2.2.2.6 Equality and Type Families

If we take the equality type as built into our theory, rather than defined as an inductive family, then the different return types of each constructor can be simulated by requiring each constructor to take as arguments proofs that the return parameters are equal to the desired values for that constructor$^7$. For example, we could formulate vectors as follows:

```haskell
data Vec’ (X : Type) (n : Nat) : Type where
    Nil’ : (y : Nat) → (y =_N 0) → Vec’ X y
    Cons’ : (y : Nat) → (n : Nat) → X → Vec’ X n → (y =_N Suc n) → Vec’ X y
```

In both constructors, the return type is $\text{Vec’} X n$, and the equality proof arguments ensure that we can only provide 0 as the $n$ value for $\text{Nil’}$, with a similar constraint for $\text{Cons’}$. So long as the eliminator for $\text{Vec’}$ is allowed to vary in the values of $n$ which the eliminator can access recursively, this type is equivalent to the normal indexed definition of $\text{Vec}$. Programming in this style is cumbersome, but because

$^6$In some presentations of type theory, transport behaves like our subst and a coercion function coe converts between equal types. Our naming convention is chosen to align with the Agda cubical library, as well as to avoid confusion between gradual casts and dependently typed coercions.

$^7$In Programming Languages folklore, this practice is known as “Fording”, in reference to the Henry Ford quip that customers could have cars in any colour, so long as it was black.
using it in a core language nicely separates reasoning about sum types (via pattern matching) and constraints on indices (by eliminating equality proofs).

### 2.2.2.7 Empty and Unit

By defining an inductive type with no constructors, we obtain an empty type, the Curry-Howard analogue to falsehood:

```haskell
data Empty : Type where  
  (no data constructors)
emptyElim : (P : Empty → Typeℓ) → (z : Empty) → P z
```

Because the type has no constructors, the eliminator takes no input other than the motive and the value itself, letting us produce a value for any motive type from a value of `Empty`. From this eliminator, we obtain the principle of “reduction to absurdity” in logic: we can derive an inhabitant of any type (i.e. a proof of any statement) from falsehood.

No closed inhabitants `Empty` exist, so any contexts where a proof of `Empty` is in scope are unreachable. The eliminator for `Empty` lets us construct explicit proofs that a particular branch of code is dead, and to use `falseElim` to avoid providing a result in those branches.

The dual of the empty type is the unit type, with a single inhabitant, and an eliminator establishing the uniqueness of that inhabitant:

```haskell
data Unit : Type where
  unit : Unit
unitElim : (P : Unit → Typeℓ) → P unit → (z : Unit) → P z
```

We sometimes refer to the empty type as `0` and the unit type as `1`, since they have zero and one elements respectively.

### 2.2.2.8 Intensional vs. Extensional Equality

Definitional equality is inherently an intensional equality: although it compares terms after normalizing, it still relies on the syntactic equality of these normal
forms. This reliance on syntax means that functions that have different implementations are not definitionally equal, even if they produce the same outputs for every input. Propositional equality is essentially the internalization of definitional equalities as first-class values in the language. \texttt{refl} can only be used to prove that two definitionally equal terms are propositionally equal, but when equality proofs are in scope, they can be used with the \texttt{J} rule to derive equalities for terms that are not necessarily definitionally equal. For example, two distinct variables \( x \) and \( z \) are not definitionally equal, but if proofs of type \( x \equiv y \) and \( y \equiv z \) are in scope then one can form a term of type \( x \equiv z \).

The most common deviation from intensionality is \textit{function extensionality}, the principle that, when two functions produce equal output for each input, they are propositionally equal. Function extensionality cannot be proved in \textsc{cic} or \textsc{mltt} without additional axioms, but it is known to be consistent to add it as an axiom to these languages. Alternately, function extensionality can be proved directly by extending the language with various features. For example, observational equality \cite{105,124} defines propositional equality by cases for each type, and equality reflection \cite{74} expands definitional equality to include any two terms for which a propositional equality can be proven in the given scope. Both of these additions imply function extensionality.

\subsection{Generic Representations of Inductive Types}

Inductive types as described above fit into an \textit{open type theory}: instead of having a fixed set of type constructors, the user can define their own. Open type theories are useful for the programmer, but proving facts about such a system is difficult, since we must reason about every possible combination of type definitions.

Here, we present ways that inductive definitions can be defined in a \textit{closed type theory}, using a fixed set of type formers. A closed type theory is particularly useful for Chapter 6, where we prove that approximate normalization terminates, even with inductive types.
2.2.3.1 Functors and Fixed Points

The key property of inductive types is their recursive structure. Each inductive type $T$ corresponds to a function $F : \text{Type} \to \text{Type}$ such that $T$ is isomorphic to $F \, T$. That is, $T$ is a fixed-point of $F$.$^8$ This function $F$ is called a functor, and each inductive type is the least fixed-point of some functor, which can generally be obtained by replacing self-reference in the definition of $T$ with the bound variable $X$ of $F$. The strict-positivity condition for inductive types is a syntactic over-approximation ensuring that the functor for each inductive type is a functor in the category-theory sense, that is, for all types $S_1, S_2$ one can define a well-behaved function $\text{map} : (S_1 \to S_2) \to F \, S_1 \to F \, S_2$.

Unfortunately, most type theories are not strong enough to construct the fixed-point of an arbitrary functor, even given proof that it is strictly positive. Instead, we must use syntactic descriptions from which functors and their fixed-points can be computed. The following sections describe two techniques for doing this. We use both of these techniques in Chapter 6 when devising a syntactic model of a gradual language with inductive types.

2.2.3.2 Containers and W-types

Containers provide a syntactic way to describe functors [1]. A container is a pair $(S, P)$ with type $(X : \text{Type}) \times (X \to \text{Type})$, where $S$ and $P$ are called the shape and position respectively. The functor $\lambda(S, P)$ described by $(S, P) : \text{Type} \to \text{Type}$ is given by $\lambda X.((x : S) \times (P \, x \to X))$.

Well-founded trees, commonly called W-types, are the fixed-points of functors described by containers. W-types are often added as primitives to a base type theory, like MLTT, but for clarity, we write them using the notation of inductive

---

$^8$Technically these are not true fixed-points, since they are only up to isomorphism, but in univalent type theories [149] these isomorphisms can be made into equalities to obtain true fixed-points.
types:

```haskell
data W (S : Type) (P : S → Type) where
    Wsup : (x : S) → (P x → W S P) → W S P
wElim : (X : W S P → Type)
    → ((s : S) → (f : (P s) → W S P) → ((p : P s) → X (f p)) → X (sup s f))
    → (x : W S P) → X x
```

The constructor `Wsup` takes a (curried) argument of type `[(S, P)](W S P)`, and since `Wsup` is the only way to construct an element of `W S P`, then `W S P` is equivalent to `[(S, P)](W S P)`, that is, it is a fixed point of the functor for a container. The eliminator provides a generalization of induction: we can derive something for each value of a `W`-type if, for each shape and position function, given an instance of the motive for each recursive value at each position (i.e. each “smaller” value), we can find an instance of the motive for `Wsup` of that shape and position function.

`W` can be used to create a type equivalent to some inductive type. The intuition is that the shape `S` should be a dependent pair, where the first element is a tag corresponding to a constructor of the inductive type, and the second element contains the arguments to that constructor, except for any recursive references to the type itself. The position then encodes how many recursive references to the type occur, or more generally, for which types `Y` the given constructor has a field `Y → T`.

The trick to devising the position is to think “logarithmically”, under the intuition that a function type `A → B` can be written as `B^A`, and will obey the standard properties of exponents in arithmetic in relation to sums and products. If a constructor for `T` has type `(a : A) → B(T, a) → T`, then `A` is the shape type for this constructor, and finding the position for that constructor amounts to finding `P` such that `T^P` is equivalent to `B(T, a)`. Products in a constructor’s fields become sums in the position, `1` becomes `0`, etc.

To make this idea concrete, consider the task of finding a `W`-type representa-
tion of \texttt{List A}. We need to find \( P_{\text{nil}} \) such that \((\text{List A})^{P_{\text{nil}}}\) is equivalent to \( \mathbb{1} \) (since \texttt{nil} has no list arguments), and \( P_{\text{cons}}(a) \) such that \((\text{List A})^{P_{\text{cons}}(a)}\) is equivalent to \texttt{List A}. Thinking logarithmically, \((\text{List A})^{0}\) is \( \mathbb{1} \) and \((\text{List A})^{1}\) is \((\text{List A})\). The result is:

\[
W (\mathbb{1} + A) (\text{sumCase} (\lambda x. 0) (\lambda x. 1))
\]

In the \texttt{nil} case, the shape argument is \texttt{inj\_1 unit}, so the remaining data has type is \( \mathbb{0} \rightarrow \text{List A} \), which is equivalent to \( \mathbb{1} \), that is, the constructor takes no arguments of type \texttt{List A}. In the \texttt{cons} case, the shape argument is \texttt{inj\_2 a} for some \( a : A \), so the remaining data has type \( \mathbb{1} \rightarrow \text{List A} \), which is equivalent to \texttt{List A}. The type of binary trees with data at each node can be obtained by replacing \( \mathbb{1} \) with \( B \) in the position, since non-leaf nodes contain two subtrees.

Without function extensionality, the straightforward encoding of an inductive type as a \( W \)-type is generally not as powerful as the original inductive type. However, Hugunin [79] recently discovered an alternate formulation of inductive types as \( W \)-types which does not require extensionality to produce equivalent types.

An indexed version of containers and \( W \)-types also exists. The basic idea is that the shape-type \( S \) should be parameterized over whatever type \( J \) indexes the inductive type, and the position provides a type for each \( i : J \) and \( s : S i \). Indexed containers contain an additional “next” field which describes, for each recursive reference in the codomain of the position, what the value of the index should be for that reference.

### 2.2.3.3 Descriptions

Descriptions give a less compact but more direct alternative to containers and \( W \)-types for encoding functors and inductive types. Many presentations of descriptions exist [29, 52, 97], but for this dissertation, we follow the presentation
of Diehl and Sheard [48]:

```plaintext
data Desc : Type, where
  End : Desc
  Arg : (A : Type) → (A → Desc) → Desc
  Rec : (A : Type) → Desc → Desc
```

Descriptions have a tree-like structure: each node either describes a recursive (Rec) or non-recursive (Arg) part of an inductive type, where branches are terminated with End. The Arg constructor serves two purposes. First, it can be used to encode a description for each constructor of a type, for example, by setting A to be some finite type of tags. It also encodes non-recursive fields of constructors, enabling dependent products by giving a description of what data should be present for each possible argument of type A. Rec describes what fields of the form A → T are contained in constructors for T.

The functor described by a description is given by the El function:

```plaintext
El : Desc → Type → Type
El End X = 1
El (Arg A D) X = (a : A) × El (D a) X
El (Rec A D) X = (A → X) × El D X
```

That is, End takes no data, Rec takes a function from A to the input type X plus whatever is encoded in the rest of the description, and Arg takes a dependent pair of a : A plus data specified for whatever description D gives for a.

The fixed point of a description can then be given as an inductive type, which closes the loop and turns each use of X in El into a recursive reference:

```plaintext
data μ (D : Desc) : Type where
  init : El D (μ D) → μ D
```
For example, the description-encoding for List A would be:

$$
\mu (\text{Arg } B(\lambda b. \text{if } b \text{ End } (\text{Arg } A (\lambda x. \text{Rec } 1 \text{ End}))))
$$

Here, B denotes the type of booleans. The first piece of data in a List is a boolean describing whether it is a nil or a cons. In the nil case, we take no more data, marked by End. In the cons case, we take a head value of type A, followed by a tail value of type 1 \rightarrow List A, which is equivalent to List A.

Descriptions do not rely on extensionality for encoding datatypes directly. Moreover, they are easier to inspect than W-types. Containers are function-oriented, making it difficult to introspect on the structure of an inductive type from its equivalent container. Descriptions, on the other hand, are more data-oriented than function-oriented, so it is generally easier to inspect the structure of a description and recover the structure of the original inductive data type.

Like W-types, descriptions have an indexed variant. With an index type I, El has type (I \rightarrow Type) \rightarrow I \rightarrow Type. The idea is that the End constructor takes an argument specifying an index value, and El for End takes a propositional equality proof that the given index matches the one specified in the description. Likewise, Rec has an additional argument specifying an index value for each recursive reference.

### 2.2.4 Dependent Types: Implementation and Practical Details

Because of the type-term overlap, implementing a dependently typed language brings with it unique challenges. Here, we highlight those that are particularly relevant when mixing gradual and dependent types, along with standard techniques for dealing with them in non-gradual settings.

#### 2.2.4.1 Local Inference and Bidirectional Type Checking

In lambda calculi, the desired type of an expression is not always clear. Even in the simply-typed lambda calculus, a term such as \( \lambda x. x \) may be assigned infinitely many types. The situation is more complex with dependent types: even if we know the type family a term should belong to, we may not be able to infer the
parameters or indices of that family from the term alone.

One particularly elegant solution to this problem is bidirectional typing [123]. Instead of a single typing judgment \( \Gamma \vdash t : T \), a bidirectional system has two judgments. The checking judgment, \( \Gamma \vdash t \leftarrow T \), treats the type of an expression as input, while the synthesis judgment, \( \Gamma \vdash t \Rightarrow T \), treats the type as output. Bidirectional typing does not remove the need for type ascriptions, but it allows typing information from top-level ascriptions to be propagated into sub-terms, meaning that in most cases, only top-level function definitions need annotations.

In most bidirectional systems, the switching between checking and synthesis is mediated by two generic rules. First, a type-ascribed expression synthesizes the type in its annotation, and checks its body against that type.

\[
\text{Ascr} \quad \frac{\Gamma \vdash t \leftarrow T}{\Gamma \vdash (t :: T) \Rightarrow T}
\]

In the other direction, we let any term check against the type it synthesizes:

\[
\text{CheckSynth} \quad \frac{\Gamma \vdash t \Rightarrow T' \quad T' = T}{\Gamma \vdash t \leftarrow T}
\]

The equality between \( T \) and \( T' \) can be replaced by a more permissive relation, such as subtyping or, as we do in this thesis, gradual consistency.

Bidirectional typing helps pinpoint where type ascriptions are needed to enable checking. In particular, function definitions do not synthesize types, so they must either be annotated, or used in contexts where their desired type is known. By using a bidirectional system, the implementation of a type checker may closely match the typing rules, since it removes the need for the rules to nondeterministically “guess” the desired type for an expression. Löh et al. [89] provide a reference implementation of a simple dependently typed language using bidirectional typing.

Lennon-Bertrand [86] presents the Bidirectional Calculus of Inductive Constructions (bCIC), which uses an alternate style of bidirectional typing. In bCIC, functions are still ascribed with their argument types, and all terms are synthesizing terms. Some typing rules have premises that are checking rules, in cases where the desired type of an expression is known. For example, when synthesize-
ing a type for a function application, the argument is checked against the function
domain type, since it is already known from synthesizing the function’s type. BCIC
has a single checking rule, which says that a term checks against any type that is
definitionally equal to the type it synthesizes.

BCIC is equivalent to CIC, but the proof relies on the transitivity of definitiona
equality. The reliance on transitivity provides insight into why bidirectional typ-
ing is preferable to, or at least, not equivalent to, unidirectional typing for gradual
languages, where the consistency relation used in place of definitiona equality is
not transitive. We use BCIC as the starting static language for Chapters 5 and 6.

2.2.4.2 Compile-time Normalization

In the CHECKSYNTH rule above, the type-term overlap complicates the issue of
what it means for two dependent types to be equal. For example, are \text{Vec Nat} 2
and \text{Vec Nat} (1 + 1) the same type? If we compare them syntactically, they are
different, but they are computations resulting in the same type.

To account for this, instead of simple syntactic equality, dependent types use a
notion of definitiona equality \cite{93}, which we write as \(=_{\alpha\beta\eta}\). Two terms are definitiona-
ionally equal if they are \(\alpha\)-equivalent after being fully \(\beta\)-reduced and \(\eta\)-expanded.
So our CHECKSYNTH rule would become:

\[
\frac{\Gamma \vdash t \Rightarrow T'}{\Gamma \vdash t \Leftarrow T'}
\]

\(T' =_{\alpha\beta\eta} T\)

The main way to implement this rule to normalize the terms involved: two terms
are definitionally equal if and only if we can reduce them to the same normal
form.

Normalization is similar to, but not identical to, run-time evaluation. Typically,
run-time evaluation is only defined for closed terms, and evaluation never
takes place under binders. The body of a function is not evaluated until that func-
tion is applied. When comparing types, we want to consider \(\lambda x. x - (1 + 1)\) and
\(\lambda x. x - 2\) to be the same. So we need to normalize under binders (\(\lambda\), \(\rightarrow\) and \(\times\)),
meaning that we need to know how to normalize terms with free variables.

Normalization under binders is achieved by defining normal forms as mutually
inductive with neutral terms. A neutral term is one of the form \(x \cdot \vec{p}\): it has a
variable, called the head, and a spine [26], which is a list of eliminators \( \overline{p} \) applied to the head. An eliminator could be an argument (for function application), a pair projection, or the eliminator for an inductive type. This convention lets us exclude terms like \((\lambda x. x) y\) from our normal forms, since it can be further reduced, while allowing \(\lambda x. (x y)\), which cannot be further reduced. A term is then in normal form if it is neutral, it is a constructor applied to normal forms, or it is a function with a normal body.

A variant of this approach is to use canonical forms [72, 121]. Instead of normalizing at each application of CHECKSYNTH, the canonical-forms approach only considers normal form types in checking and synthesis, and allows any term to check against the type it synthesizes. Since judgments only refer to normal forms of types, a notion of hereditary substitution [154] is required for function application and other forms with types depending on values. In hereditary substitution, substitutions performed on normal forms are further reduced to produce normal forms as a result. Neutral forms in hereditary substitution are also called atomic forms. Hereditary substitution is particularly elegant, because in a calculus without inductive types, normalization can be written as a structurally-recursive procedure. We use this in Chapter 3 to prove the termination of approximate normalization for gradual types.

2.2.4.3 Implicits, Unification, and Type Inference

Dependent types as we have presented them are incredibly cumbersome to use. For example, the eliminator for vectors takes several arguments, and requires that the motive be manually specified. Likewise, refl requires one to explicitly provide the \( t \) for which \( t = t \) is being proved, along with the type of \( t \).

In practice, dependently typed languages alleviate this verbosity through a system of program metavariables and implicit arguments. A program metavariable, usually written as an underscore \( _\_ \), is a special syntactic construct that informs the compiler to infer what value must replace the metavariable for the program to be type correct. For example, in \texttt{refl} \( _3 \), \texttt{Nat} is the only valid type value, since \( 3 \) has a known type. Implicit arguments in a function declaration, written using curly-braces \( \{ x : X \} \), form a shorthand that instructs the compiler to insert a
program metavariable for that function argument unless it is explicitly provided.

Though dependent types do not have a separate construct for type polymorphism, the fact that we can pass types as arguments lets us use the normal $\lambda$ construct similarly to the $\Lambda$ binder of System-F [125]. For example, we can write $(\lambda X \cdot x) : (X : \text{Type}_0) \to X \to X$ for an explicitly polymorphic identity function. With implicit arguments, we can eliminate the need for explicitly instantiating polymorphic functions. If instead we write $id = (\lambda X \cdot x) : \{X : \text{Type}_0\} \to X \to X$, we can call $id t$ for any term $t : T$, and let the compiler determine that $T$ should be provided for the parameter $X$ letting us write polymorphic code in a style reminiscent of Haskell or ML.

Program metavariables must be resolved using unification [109]. Unlike the inference system of Damas and Milner [44], however, this inference cannot be complete. The type-term overlap, and the fact that dependent functions and pairs have binding structure, means that we must use higher order unification [77] to solve our metavariables. Higher order unification is undecidable in the general case [78], although it has a decidable subproblem called the pattern fragment [100]. This fragment is large enough that it can be used to solve most unification problems that arise with dependent types [3, 71].

Type inference has an interesting relationship with gradual typing because of the subtle difference between a type with a single static value that is not yet known and a gradual type that can potentially represent many static types. The consistency relation between types resembles unification, with each ? occurrence acting as a distinct metavariable. Siek and Vachharajani [134] provide an account of mixing type inference and gradual typing without dependent types, as do Garcia and Cimini [66]. We discuss possible future work on gradual higher order unification in Chapter 7.

### 2.2.5 Syntactic Models of Dependently Typed Languages

Proving termination is one of the main strategies for proving consistency of a language: the usual argument combines the fact that the empty type contains no normal form terms with the fact that all terms eventually reduce to a normal form. Conventional decreasing-metric arguments are typically difficult, if not im-
possible, to use when showing that all terms in a dependently typed language terminate.

The common approach for showing termination of a system is to construct a *model* of the language: devise some mathematical structure for each type, where each term of that type can be mapped to an instance of that structure called a *denotation*, and that the denotation of a term is invariant under reduction. Then, if the empty type has an empty denotation, there can be no closed terms of that type. Models of dependently typed languages are often based on partial equivalence relations or category theory [51].

A simpler form of model is a *syntactic model* [11, 17], where instead of having an abstract mathematical denotation, each term’s denotation is a term from a dependent type theory. Syntactic models let us “piggyback” off of a model for an existing type theory, such as MLTT or CIC. Depending on how the model is constructed, we can then use properties of the target type theory to prove facts about the source type theory. For example, if we give a language a syntactic model in a strongly normalizing type theory, and each source redex has a denotation containing a target-language redex, then strong normalization for the source language follows from a straightforward simulation argument. Likewise, consistency can be shown by ensuring that empty source types translate to empty types in the syntactic model.

Syntactic models are well suited to showing weaker forms of consistency for dependently typed languages that lack full consistency. For example, if the type *Empty* has a denotation that is a target singleton type, then there is a unique characterization of what source terms can inhabit *Empty*. This technique is used, for example, to model that errors are the only inhabitants of empty types in a type theory with exceptions [118, 120], and to show that errors and *?* are the only inhabitants of empty types in a gradual type theory [87, 90].

### 2.3 Prior Work on Gradual Dependent Types

Here, we describe existing work in the space of mixing dynamic and dependently typed code. To our knowledge, our work [58], on which Chapter 3 is based, is the first to address *fully gradual* dependent types, where any type or term can be
replaced by ?.

2.3.1 Contracts and Dynamic Checking

Findler and Felleisen [65] introduced higher order contracts, which provide a way for programmers to specify properties of functional programs. Among their contributions are dependent contracts, which allow the contract on a function’s return value to depend on the value of the argument. Though dependent contracts provide a form of dynamic dependent typing, they are purely dynamic, offering no static guarantees on code.

Ou et al. [114] provide a system in which static dependent types are mixed with dynamic checks, in one of the earliest attempts to bring gradual typing concepts to dependent types. However, their language is oriented around subset types, which constrain a simple type to only those values satisfying some boolean predicate. The reliance on boolean predicates makes it easy to move compile-time checks to run time. Boolean predicates are strictly less expressive than full dependent types, and are not compatible with much of the existing dependently typed code. Any proofs of theorems that are not of the form \( p(x) \equiv \text{true} \) cannot be omitted by the user.

Tanter and Tabareau [144] modify Coq, a dependently typed theorem prover, to have dynamic enforcement of type constraints. Using a combination of unsound axioms and Coq’s typeclass and implicit coercion mechanisms, they allow for a value \( t : T \) to be automatically converted into the type \((t : T) \times (p(t) \equiv \text{true})\), with the check that \( p \) holds dynamically performed. This work also relies on boolean predicates and subset types, and doesn’t allow for arbitrary proofs to be omitted.

Gradual refinement types provide a limited form of term-dependence in types. Lehmann and Tanter [84] provide a system for gradual refinement types, lightweight dependent types providing predicates over numbers. This work provides many of the same properties as our proposed language, such as the ability to smoothly move between more or less precise types. However, the techniques used for refinement types do not scale to full dependent types. Zalewski et al. [158] present a blame-safe system for gradual refinement types with dependent functions, but
achieve decidable type checking through a strict distinction between terms and types.

2.3.2 Gradual Type Theories

Several attempts have been made to incorporate concepts from gradual typing into formal type theories, with various levels of compromise on consistency, convenience, and the gradual guarantees.

Osera et al. [113] present dependent interoperability, a system for interfacing between a (simple) static and dependently typed language. The approach relies on a one-to-one correspondence between constructors of types in the static and dependent languages, and requires users to provide their own functions for converting between the two. This technique was extended by Dagand et al. [42, 43] to use partial equivalences, a loosening of equivalences from type theory. These equivalences take a way to map a simple type, such as List T into a dependent type, like Vec T n and back, with the former having the possibility of failure. Given such an equivalence, functions operating on one type can be automatically lifted to work with the other. Type failure is expressed using an error monad. The partial equivalence approach works with any types that can be rewritten to be expressed as a subset type.

Dependent interoperability is related to gradual typing. However, its focus is on interactions between simply-typed and dependently typed code, where we seek to support the whole spectrum between dynamic and dependently typed.

Exceptional type theory provides a formalization of dependently typed languages with catchable run-time exceptions [118, 120]. These languages do not support gradual typing. Moreover, gradual type errors are traditionally not catchable, as catching errors violates the gradual guarantees. However, the ability to formally model terms that may fail provides valuable insights that can be used when describing the behaviour of approximate normalization. Pédrot and Tabareau [119] show that dependent elimination and substitution must be restricted in the presence of observable effects, but the non-catchable run-time type errors of gradual typing do not fall into this category.

Lennon-Bertrand et al. [87] combine aspects of gradual typing and exceptional
type theory to provide a fully gradual Calculus of Inductive Constructions (CIC). They identify three properties which cannot possibly co-exist in a gradual dependently typed language: conservatively extending the static sub-language (CIC in their case), strong normalization, and the embedding-projection (EP) property. The EP property, first used in gradual typing by New and Ahmed [105], is the property that losing and re-gaining precision on a term produces the original term, and gaining then losing precision on a term produces the same term, or an error.

The Gradual Calculus of Inductive Constructions (gCIC) is presented as three language variants, based on a common framework, that fulfill each combination of the above properties. The work on gCIC was published after our initial publication [58], and inherits some features from that work, such as the use of \( ? \) as both a term and a type.

gCIC was extended to a “reasonably gradual” type theory (GRIP) by Maillard et al. [90]. GRIP sacrifices the full gradual guarantees, but identifies the fragment of their language that satisfies the gradual guarantees. The type theory is divided into two layers, a gradual layer in which all terms may fail or produce \( ? \), and a propositional layer which can be used to reason about terms in the gradual layer. Notably, gradual precision is reflected into the propositional layer, so the precision relation between terms can be established in the type theory itself.

2.3.3 Dependently Typed Non-termination

2.3.3.1 Allowing Non-terminating Functions

We are not the first to encounter the problem of running possibly non-terminating during dependent type checking. The Zombie language [23, 138] featured two distinct sub-languages. One was logically consistent and could be used for theorem proving, while the other featured general recursion and was used for programming. Though no false theorems can be proved in such a language, the termination of type checking is not guaranteed. In many ways, their language is the converse of what we present in Chapter 3: they have two separate sub-languages with identical semantics but limited interactions, while we want a single language that has different behaviour depending on whether it is run at compile time or run time.

Other approaches to non-termination include that of Dependent Haskell [55],
where no termination checks are performed at compile time and logical consistency is lost, or Idris [19], where the user must declare which functions are total and may be executed at compile time.\(^9\)

### 2.3.3.2 Modelling Non-termination

A separate line of research involves using dependently typed languages to *model* non-terminating computations. Such systems have the ability to *describe* possibly infinite computation, while keeping all functions in the core language terminating. The advantage of this approach is that minimal changes are required to the type theory itself, and the programmer can rest assured that functions they write in said type theory always terminate, as does type checking. The disadvantage is that fewer definitional equalities are present: since non-terminating procedures are defined as objects describing functions, rather than functions themselves, the normal $\beta$-equalities are not automatically applied during type checking.

Most approaches to modelling non-termination use some sort of modality, possibly a monad. Capretta [21] uses a coinductive type, commonly called the *delay monad*, to model general recursion. McBride [98] generalizes this technique using a *free monad*, where a generic interface is provided for defining general-recursive functions. Semantics may then be given to this interface using one of several methods, such as a fuel-based semantics, or using a delay monad (in an approach adapted from Abel and Chapman [2]).

*Guarded recursion* [103] is particularly useful for gradual types, since it can describe non-terminating computation and non-positive fixed points using a *later* modality, where data under the modality is only available at some time in the future. The later modality can be used to express many coinductive computations [7]. Guarded type theory can be given a model using the *topos of trees* [14], and provides a type-theoretical analogue to step-indexing in logical relations. Several dependent type theories have been developed based on guardedness [13, 16], along with various computational interpretations [15, 140]. In Chapter 6, we use guarded recursion to model the non-terminating parts of gradual run-time computations.

---

\(^9\)At the time of writing, current implementation of Idris 2 allows non-termination at compile time, but adding the check from Idris 1 is planned to eventually be added back to Idris 2.
Chapter 3

Foundations of Gradual Dependent Types

This chapter presents a Gradual Dependently Typed Language (GDTL), which originally appeared in [58]. GDTL is a foundational gradual dependently typed language: at the time of publication, to our knowledge, it was the only calculus in the literature that enabled \( ? \) to be used as both a type and a term.

Our aims for this chapter are twofold. First, we make a first foray into the design space for gradual dependent types. With GDTL, we identify the key ways that gradual dependent types should differ from gradual simple types. Specifically, the dependence of types on terms means that \( ? \) should be allowed to be used as both a term and a type, so that the programmer can express types with a known type constructor but imprecise indices. Likewise, proof terms in dependently typed languages can persist at run time and even affect program behaviour, so \( ? \) should be allowed to replace proof terms, and should have defined run-time semantics. We adopt the approach of Abstracting Gradual Typing (AGT) to ensure that gradual capabilities are added to dependent types in a safe, principled way.

The second task for this chapter is to make precise the relationship between gradual types, dependent types, and computational effects. Specifically, gradual types introduce both non-termination and dynamic type errors, which are in tension with the need to normalize terms at compile time when checking dependent types. To account for non-termination and failure, we distinguish between
compile-time normalization and run-time execution: compile-time normalization is approximate but terminating, while run-time execution is exact, but may diverge. We prove that GDTL has decidable type checking and satisfies all the expected properties of gradual languages. In particular, GDTL satisfies the gradual guarantees: reducing type precision preserves typedness, and altering type precision does not change program behaviour outside of dynamic type failures. To prove these properties, we establish a novel normalization gradual guarantee that captures the monotonicity of approximate normalization with respect to imprecision.

The original publication of GDTL claimed that it satisfied the gradual guarantee, but unfortunately, there was an error in the proof. As such, we have modified the presentation of GDTL to ensure that it does satisfy the static gradual guarantees. Specifically, we include a notion of evidence on neutral terms when normalizing during type checking, storing the most precise information available about what their actual types should be. This approach lets us monotonically handle type errors.

In general, the approaches taken to proving the static guarantee in Chapters 5 and 6 are preferable to this one: the cast calculus paradigm greatly simplifies the issue of normalizing types and comparing them during type checking. Moreover, in contrast to GDTL, the languages of those chapters respect static equivalences: extensionally-equal static programs cannot be distinguished gradually. Unlike this chapter, Chapters 5 and 6 do not use a systematic approach to derive the gradual type system, but we discuss possible future work to re-systematize our approach in Chapter 7.

3.1 Introduction

A major barrier to the integration of gradual dependent types is the tension between the purity of dependently typed code and the impurity of gradually typed code. A dependent type checker must evaluate some program terms as part of type checking, but gradual types complicate this in two ways. First, if a gradual language fully embeds an untyped language, then some programs diverge: indeed, self application ($\lambda x : ? . x x$) is typeable in such a language. Second, gradual
languages introduce the possibility of type errors that are uncovered as a term is evaluated: applying the function \((\lambda x : ? \cdot x + 1)\) may or may not fail, depending on whether its argument can actually be used as a number. For example, using \(true : \?) as an argument fails, but \(0 : \?) succeeds. So a gradual dependently typed language must account for the potential of non-termination and failure during type checking.

This chapter attempts to rectify the pure-impure tension between dependent and gradual types. We present a Gradual Dependently Typed Language (\(\text{g.d.tl}\)) that supports the whole spectrum between an untyped functional language and a dependently typed one. As such, \(\text{g.d.tl}\) adopts a unified term and type language, meaning that the unknown type \(\?) is also a valid term. Having \(\?) as a term lets programmers specify types with imprecise indices and replace proof terms with \(\?) (Section 3.2).

\(\text{g.d.tl}\) is a gradual version of the Calculus of Constructions with a predicative universe hierarchy (\(\text{c.c}_\omega\)) (Section 3.3), similar to the core language of Idris [19]. We gradualize this language following the Abstracting Gradual Typing (\(\text{a.g.t}\)) methodology [67] in Section 3.4. Because \(\text{g.d.tl}\) is a conservative extension of this dependently typed calculus, it is both strongly normalizing and logically consistent for fully static code. However, these strong properties are lost as soon as imprecise types and/or terms are introduced. On the dynamic side, \(\text{g.d.tl}\) can fully embed the untyped lambda calculus. When writing purely untyped code, static type errors are never encountered. In between, \(\text{g.d.tl}\) satisfies the static and dynamic gradual guarantees of Siek et al. [136], meaning that evaluation is monotone with respect to type imprecision. These guarantees enable the programmer to move between imprecise and precise types in small, incremental steps, with the program type checking and behaving identically (modulo dynamic type errors) at each step.

The key technical insight on which \(\text{g.d.tl}\) is built is to exploit two distinct notions of evaluation: one for normalization during type checking, and one for execution at run time. Specifically, we present a novel semantics, called approximate normalization. Approximate normalization is not used to define the runtime behavior of programs, but rather, as a means of defining an approximate notion of definitional equality that determines whether two types are equivalent
in the eyes of the type checker. Approximate normalization allows decidable type checking (Section 3.5): applying a function of unknown type, which may trigger non-termination, normalizes to the unknown value ?. Consequently, some terms that would be distinct at runtime become indistinguishable as type indices. Compile-time tracking of type information during normalization safely handles “dynamic” type errors occurring during static normalization. In this sense, GDTL is closer to the styles of Idris, Agda, Coq, etc. than Dependent Haskell [55], which does admit reduction errors and non-termination during type checking. At runtime, GDTL uses the standard, precise run-time strategy of existing gradual languages, which may fail due to dynamic type errors, and may diverge as well (Section 3.6). In that respect, GDTL is closer to Dependent Haskell and Zombie [23, 138] than to Idris, which features a termination checker and a static effect system. We prove that GDTL has decidable type checking and satisfies several desirable properties of gradual languages [136]: type safety, conservative extension of the static language, embedding of the untyped language, and the gradual guarantees (Section 3.7). Section 3.8 discusses related work, and Section 3.9 discusses limitations and perspectives for future work.

3.2 Goals and Challenges

In this section, we revisit the example of Section 1.3, introducing GDTL by showing how the vector head operation behaves when given various inputs. To propagate imprecision to type indices, and soundly allow omission of proof terms, GDTL admits ? both as a type and a term. To manage effects due to gradual typing, we use separate notions of evaluation for compile time and run time. Introducing imprecision in the compile-time normalization of types avoids both non-termination and failures during type checking.

This section uses both vectors and propositional equality for the sake of exposition, so that the examples are actually comprehensible by a human. Neither of these features is directly supported by GDTL, since GDTL is a foundational calculus, and does not support inductive types. Analogous types can be defined using Church encoding, although the predicative nature of GDTL weakens how these can be used. However, we emphasize that the purpose of this chapter is to ex-
plore decidable type checking for gradual dependent type. Features necessary for practical programming are discussed in Chapter 5.

### 3.2.1 The unknown as a type index

Recall that the type of vectors has constructors \( \text{Nil} : \text{Vec} \ A \ 0 \) and \( \text{Cons} : A \rightarrow \text{Vec} \ A \ n \rightarrow \text{Vec} \ A \ (n + 1) \), and a function \( \text{head} : \text{Vec} \ A \ (1 + n) \rightarrow A \).

Since full-spectrum dependently typed languages conflate types and terms, \( \texttt{?} \) lets be used as either a term or a type. Just as any term can have type \( \? \), the term \( \? \) can have any type, allowing imprecision in indices to be specified. For example, we can define vectors \( \text{staticNil} \), \( \text{dynNil} \) and \( \text{dynCons} \) as follows:

\[
\begin{align*}
\text{staticNil} & : \text{Vec} \ Nat \ 0 \\
\text{dynNil} & : \text{Vec} \ Nat \ ? \\
\text{dynCons} & : \text{Vec} \ Nat \ ? \\
\text{staticNil} & = \text{Nil} \\
\text{dynNil} & = \text{Nil} \\
\text{dynCons} & = \text{Cons} \ 0 \ (\text{Nil})
\end{align*}
\]

Then:

- \( \text{head staticNil} \) does not type check,
- \( \text{head dynNil} \) type checks but fails at run time,
- \( \text{head dynCons} \) type checks and succeeds at run time

The programmer can choose between compile time or run time, but safety is maintained either way, and in the fully-static case, the unsafe code is still rejected.

### 3.2.2 The unknown as a term at run time

Having \( \? \) as a term means that programmers can use it to optimistically omit proof terms. Indeed, terms can be used not only as type indices, but also as proofs of propositions. For example, we can use propositional equality to write a (slightly contrived) formulation of the \( \text{head} \) function:

\[
\text{head'} : (n : \text{Nat}) \rightarrow (m : \text{Nat}) \rightarrow n =_{\text{Nat}} m + 1 \rightarrow \text{Vec} \ A \ n \rightarrow A
\]

This variant accepts vectors of any length, provided the user also supplies a proof that its length \( n \) is not zero by explicitly providing the predecessor \( m \) and the
equality proof of type $n =_{\mathbb{N}} (m + 1)$. GDTL lets ? be used in place of a proof, while still ensuring that a run-time exception is thrown if $head'$ is ever given an empty list. For instance, suppose we define a singleton vector and a proof that $0 =_{\mathbb{N}} 0$:

\[
\begin{align*}
\text{staticCons} & : \text{Vec Nat} 1 \\
\text{staticProof} & : 0 =_{\mathbb{N}} 0 \\
\text{staticCons} & = \text{Cons Nat} 0 (\text{Nil Nat}) \\
\text{staticProof} & = \text{Refl Nat} 0
\end{align*}
\]

Then:

- $head' \text{Nat} 0 0 \text{staticProof} \text{staticNil}$ does not type check
- $head' \text{Nat} 0 ?? \text{staticNil}$ type checks but fails at run time
- $head' \text{Nat} 1 ?? \text{staticCons}$ type checks and succeeds at run time

We defer discussion of the specifics of gradual propositional equality to Chapter 5, where we develop a more sophisticated method of tracking run-time equality information. The important point is that, because dependent types use proofs that can persist at run time, ? should be treated as a term with properly defined dynamic semantics.

### 3.2.3 Managing effects from gradual typing

To illustrate how the effects of gradual typing can show up in type checking, suppose a programmer uses ? to write a gradually typed Z-combinator, the call-by-value counterpart of the Y-combinator:

\[
Z = \lambda (f : ?). (\lambda (x : ?). f (\lambda (v : ?). x x v)) (\lambda (x : ?). f (\lambda (v : ?). x x v))
\]

They may then use this combinator to accidentally write a non-terminating function $badFact$.

\[
badFact = \lambda m . Z (\lambda . \text{ifzero} m (f 1)(m + f (m))) \quad -- \text{never terminates}
\]
Many languages allow non-termination, but the issue is that computing the return type of a function application may diverge. For example:

\[ \text{repeat} : (A : \text{Type}) \rightarrow (n : \text{Nat}) \rightarrow A \rightarrow (\text{Vec } A \ n) \]

\[ \text{factList} = \text{repeat } \text{Nat} \ (\text{badFact } 1) \ 0 \quad -- \text{has type } \text{Vec } \text{Nat} \ (\text{badFact } 1) \]

It turns out that any diverging code necessarily applies a function of type \(?\). Though \text{badFact} does not have type \(?\), its definition uses \(Z\), which contains ascriptions of type \(?\).

Similarly, a naïve approach to gradual dependent types encounters failures when normalizing some terms. Let \(::\) denote a type ascription in the surface language, which can induce an implicit type conversion. How should we attempt to type check the following term?

\[ \text{failList} = \text{repeat } \text{Nat} \ (\text{false }::?) \ 0 \]

The return type of this call is \(\text{Vec } \text{Nat} \ (\text{false }::?)\), that is, a vector with length \text{false}. It is fairly clear that such a nonsensical example should not type check, since normalizing \text{false }::? at type \text{Nat} should produce an error. However, not all dynamic errors are this straightforward. Consider a function that gives \text{repeat} a length of unknown type:

\[ \text{failFun} = \lambda(x : ?). \text{repeat } \text{Nat} \ x \ 0 \]

Just because a function raises a type error on some inputs does not mean that normalizing it should fail. So we need a notion of compile-time normalization that can raise a type error for \text{failList}, but is lazy enough to let \text{failFun} be called on \(?\)-typed inputs that normalize to numbers.

\text{GDTL} avoids both problems by using different notions of running programs for the compile time and run time. We distinguish compile-time normalization, which is approximate but terminating, from run-time execution, which is exact but may diverge. When non-termination is possible, compile-time normalization uses \(?\) as an approximate but pure result. To account for possible failure under binders, neutral terms are ascribed with compile-time evidence, a novel version of run-time evidence from \text{AGT}, that stores type information under binders to potentially raise
an error if an invalid input is provided. So `factList` can be defined and used in run time, but are assigned type `Vec Nat ?`. Conversely, `failList` is rejected due to the error when normalizing at compile time.

GDTL normalization is based around *hereditary substitution* [154], which is a total operation from canonical forms to canonical forms. With hereditary substitution, we can define a decreasing measure in terms of the type of the value being substituted, so a static termination proof is easily adapted to GDTL. This decreasing measure lets us pinpoint exactly where gradual types introduce possible non-termination, approximating in those cases. Similarly, our use of bidirectional typing means that a single check needs to be added to prevent failures in normalization.

### 3.2.4 Gradual Guarantees for GDTL

To ensure a smooth transition between precise and imprecise typing, GDTL satisfies the *gradual guarantee* of Section 2.1.2.2. One novel insight of GDTL’s design is that the interplay between dependent type checking and program evaluation carries over to the gradual guarantees. Specifically, the static gradual guarantee fundamentally depends on a restricted variant of the dynamic gradual guarantee. We show that approximate normalization maps terms related by precision to canonical forms related by precision, thereby ensuring that reducing a term’s precision always preserves well typedness.

By satisfying the gradual typing criteria, and embedding both a fully static and a fully dynamic fragment, GDTL gives programmers freedom to move within the entire spectrum of typedness, from the safety of higher-order logic to the flexibility of dynamic languages. Furthermore, admitting `?` as a term means that we can easily combine code with dependent and non-dependent types, the midpoint between dynamic and dependent types. For example, the simple list type could be written as `List A = Vec A ?`, so lists could be given to vector-expecting code and vice-versa. The programmer knows that as long as vectors are used in vector-expecting code, no crashes can happen, and safety ensures that using a list in a vector operation always fails gracefully or run successfully. This approach is significantly different from work on casts to subset types [144] and dependent
interoperability [43], where the user must explicitly provide decidable properties or (partial) equivalences.

3.2.5 Summary of Design Decisions

GDTL embodies several important design decisions, each with trade-offs related to ease of reasoning and usability of the language.

By embracing full-spectrum dependent types, GDTL lets types be first-class citizens: arbitrary terms can appear in types and expressions can produce types as a result. The programmer does not need to learn a separate index language, and does not need to recreate term-level operations at the type level.

Sticking to clearly separated phases lets us adopt different reduction strategies for type checking and for execution. Crucially, by using approximate normalization, we ensure that type checking in GDTL always terminates: compile-time normalization is a total (though imprecise) operation. At type $\tau$, some type information is statically lost, with checks deferred to run time.

GDTL features an unknown term $\_\$, which resembles term holes in Agda and Idris, and existential variables in Coq; the notable difference is that programs containing $\_\$ can be run without evaluation getting stuck. Every type in GDTL is therefore inhabited at least by the unknown term $\_\$, so the language is inconsistent as a logic, except for fully-precise programs.

In a gradual language that can embed arbitrary untyped terms, programs may not terminate at run time. Every type in GDTL contains expressions that can fail or diverge at run time, due to imprecision. Fully-precise programs are guaranteed to terminate.

Finally, like Coq, Agda, and Idris, GDTL is based on an intensional type theory, meaning that it automatically decides definitional equality—i.e. syntactic equality up to normalization—and not propositional equality; explicit rewriting is necessary to exploit propositional equalities. Consequently, this check is reflected to run time: dynamic checks in GDTL also rely on definitional equality. Thus, equality is decidable, but a run-time error can be triggered even though two (syntactically different) terms are propositionally equal. The limitations of this approach, and alternatives to it, are described in Chapter 5.
3.3  **sDTL: A Static Dependently Typed Language**

We now present our Static Dependently Typed Language (**sDTL**), which is essentially a bidirectional, call-by-value, variant of the predicative fragment of \(\mathcal{CC}_\omega\) (i.e. the calculus of constructions with a universe hierarchy [39]). **sDTL** is the starting point of our gradualization effort.

### 3.3.1 Syntax and Dynamic Semantics

The syntax of **sDTL** is shown in Fig. 3.1, with metavariables for the static variants of terms, values, etc. written in red sans-serif font. Types and terms share a syntactic category. The syntax has functions and applications in their usual form, but function types are *dependent*: a variable name is given to the argument, and the codomain may refer to this variable. The form \(\text{Type}_\ell\) denotes a universe at level \(\ell\), i.e. a type of types. Universes have a hierarchical structure: the lowest types have the type \(\text{Type}_0\) and each \(\text{Type}_\ell\) has type \(\text{Type}_{\ell+1}\). In this chapter, we write \(\text{Type}\) to mean \(\text{Type}_0\). Finally, an explicit type ascription form \(t :: T\) lets the programmer manually specify what the type of an expression should be, which is crucial for this style of bidirectional typing.

We use metavariables \(v, V\) to range over values, the subset of terms consisting only of functions, function types and universes. For evaluation, we use a call-by-value reduction semantics (Fig. 3.1). Ascriptions are dropped when evaluating, and function applications result in (syntactic) substitution. We refer to the values and semantics as *simple* rather than *static*, since they apply equally well to an untyped calculus, albeit without the same soundness guarantees. As we see below, the semantics used to compare terms during type checking is based on hereditary substitution, and is extremely strict, even normalizing under binders. However, for the most part this is irrelevant to gradual typing: call-by-name and call-by-value cause the same static and dynamic checks to be performed, differing only in when the types involved in those checks are normalized.
3.3.2 Comparing Types: Normal Forms

sdtl compares types using normal forms. Since dependent types can contain expressions, it is possible that types may contain redexes. Most dependent type systems have a conversion rule that assigns an expression type $T_1$ if it has type $T_2$, and $T_2$ is convertible to $T_1$ through some sequence of $\beta$-conversions, $\eta$-conversions, and $\alpha$-renamings. Instead, we treat types as $\alpha\beta\eta$-equivalence classes. To compare equivalence classes, we represent them using normal forms [154], denoted with metavariables $u$ and $U$. Normal forms are $\beta$-reduced, $\eta$-long canonical members of an equivalence class. We compare terms for $\alpha\beta\eta$-equivalence by normalizing and syntactically comparing their normal forms.

Figure 3.2 defines the syntax and well-formedness (typing) for normal forms. Normal forms are either universes, functions, function types, or neutral terms tagged with their type, which must itself be either neutral or $\text{Type}_\ell$. For example, $\langle X z \rangle(y z)$ and $\langle \text{Type}_\ell \rangle(x \text{ Type}_\ell)$ are both well-formed tagged neutrals. The type tag for neutrals is not necessary for the static language, since we can synthesize their types without annotations. However, the tag simplifies the definition of the meaning of gradual types in Section 3.4. The angle-bracket notation is taken from the AGT notation for evidence, since when we gradualize sdtl, that is precisely what is affixed to neutral terms.

The definition of neutral forms is also given in Fig. 3.2. A neutral form is a variable applied to some number of normal-form arguments. Neutrals are defined
as a list-like structure, but for hereditary substitution it is convenient to always know what variable is at the end of the chain of applications. So we define neutral terms as a variable $x$ called the head, applied to a list of arguments, which we refer to as its spine [26].

The separation of neutrals and normals is key to ensuring that normal are not reducible. By only allowing applications in neutral form, we ensure that all heads are variables, and thus no redexes are present, even under binders. Well-formedness is defined bidirectionally: neutrals synthesize and normal forms check. We defer discussion of the well-formedness rules to Section 3.3.3, where we give the more general typing rules for terms. In addition to the usual type restrictions, the well-formedness rules ensure $\eta$-longness: the types of neutral forms are either neutral or $\text{Type}_\ell$ for some $\ell$, so they cannot have type $(y : U_1) \to U_2$.

### 3.3.3 Type checking and Normalization

With normal forms defined, we can present the typing rules for $\text{sdtl}$ (Fig. 3.3), where terms are assigned types in normal form. To ensure syntax directedness, we again use bidirectional typing.

The typing rules use the typical synthesis and checking judgments for bidirectional typing. The type synthesis judgment $\Gamma \vdash t \Rightarrow U$ says that $t$ has type $U$ under context $\Gamma$, where the type is treated as an output of the judgment. That is, from examining the term, we can determine its type. The checking judgment $\Gamma \vdash t \Leftarrow U$ says that, given a type $U$, we can confirm that $t$ has that type. These rules propagate the information from ascriptions inwards, so that only top-level terms and redexes need ascriptions.

The rule $\text{SSYNTHAPP}$ displays several important concepts that are key to the design of $\text{sdtl}$. Since functions are dependently typed, the rule computes the result of applying a particular term by substituting that term into the return type. First, the judgment $\Gamma \vdash u \Leftarrow t_2 \Leftarrow U_1$ computes the normal form of $t_2$ at type $U_1$. Then the return type is computed using hereditary substitution, which takes normals as inputs and produces the normal form result of replacing the variable $x$ with the argument’s normal form. Hereditary substitution, which we write $[u_1/x]^U u_2 = u_3$, is distinct from syntactic substitution, which in this chapter
we write as \( [x \mapsto t_1]t_2 = t_3 \). We explain hereditary substitution further in Section 3.3.4.

The remaining rules in the system are standard. Each universe Type, synthesizes \( \text{Type}_{\ell+1} \) (SSynthType), and variables synthesize their type from the context \( \Gamma \) (SSynthVar). The switch from checking to synthesis is mediated by the rule SSynthAnn, where \( t :: T \) checks against \( U \) so long as \( t \) synthesizes \( U \) and \( U \) is the normal form of type \( T \). We do not care what level the type \( T \) has, only that it has a normal form \( U \) at some level. The SCHECKSYNTH is the bidirectional version of a conversion check: \( t \) checks against \( U \) so long as it synthesizes a type with the same normal form \( U \). Functions are checked in the usual way: \( \lambda x. t \) checks against

\[
\begin{align*}
\text{SNF} \ni u, U & \quad ::= \text{Type}_{\ell} \mid \lambda x. u \mid \langle N \rangle N \quad (\text{Static Normal Forms}) \\
\text{SNE} \ni N & \quad ::= \pi \quad (\text{Static Neutral Forms}) \\
\text{SSpine} \ni \pi & \quad ::= \pi \mid p u \quad (\text{Static Normal Spines}) \\
\end{align*}
\]

**Figure 3.2:** sdtl: Normal and Neutral Forms
\( \Gamma \vdash t \Rightarrow U \) (Static Synthesis)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSYNTHTYPE ( \ell &gt; 0 )</td>
<td>( \Gamma \vdash \text{Type}<em>\ell \Rightarrow \text{Type}</em>{\ell+1} )</td>
</tr>
<tr>
<td>SSYNTHVAR</td>
<td>( \Gamma \vdash (x : U) \in \Gamma \Rightarrow U )</td>
</tr>
<tr>
<td>SSYNTHPAR</td>
<td>( \Gamma \vdash t \Rightarrow )</td>
</tr>
<tr>
<td>SSYNTHANN</td>
<td>( \Gamma \vdash U \leftarrow \text{Type}_\ell \Rightarrow U )</td>
</tr>
</tbody>
</table>

\[ \Gamma \vdash t \Rightarrow (x : U_1) \rightarrow U_2 \quad \Gamma \vdash u \leftarrow t_2 \Rightarrow U_1 \quad [u/x]^{U_1} U_2 = U_3 \]

\( \Gamma \vdash t_1 \Rightarrow t_2 \Rightarrow U_3 \)

\( \Gamma \vdash U \leftarrow \text{Type}_\ell \Rightarrow U \)

\( \Gamma \vdash (x : U) \leftarrow \text{Type}_\ell \Rightarrow (x : U) \Gamma \vdash t \Rightarrow U_2 \)

\( \Gamma \vdash \lambda x.t \leftarrow (x : U_1) \rightarrow U_2 \)

\( \Gamma \vdash T_1 \leftarrow \text{Type}_\ell \Rightarrow (x : U) \Gamma \vdash T_2 \Rightarrow \text{Type}_\ell \)

\( \Gamma \vdash (x : T_1) \rightarrow T_2 \leftarrow \text{Type}_\ell \)

**Figure 3.3: sDTL: Type Synthesis and Checking**

a function type if its body checks against the return type, under the assumption that its argument has the domain type. The rule SCHEKPI encodes predicative quantification: we can check function types against \( \text{Type}_\ell \), provided the domain and codomain check against \( \text{Type}_\ell \).

The normalization judgment used in SSYNTAPP is defined in Fig. 3.4, with rules directly mirroring those of well typed terms. In particular, the rule SNORMPROD \( \eta \)-expands any variables with function types, which lets us assume that the function in an application always normalizes to a \( \lambda \)-term. We use this assumption in SNORMSYNTAPP, where the normal form of an application is computed using hereditary substitution.

Auxiliary judgments supporting normalization, are defined in Fig. 3.5. The judgment \( \Gamma \vdash T \leftarrow U : \text{Type}_\ell \) works like checking against \( \text{Type}_\ell \), but produces the level \( \ell \) of the type as an output rather than taking it as an input. Synthesizing forms normalize using the normalization-synthesis judgment, discarding the
synthesized Type. Function types are the only types that are checking forms, and their level-unaware normal forms are computed by computing the normal forms for the domain and codomain at unknown level and synthesizing the maximum. The $\eta$-expansion judgment expands neutral terms into normal forms in a type-directed manner. In EtaNeut, a neutral term with a neutral type expands by adding the type as a tag, and in EtaType a neutral type expands by adding the Type tag. Finally, a neutral term with a function type expands in Etap1 by abstracting over a fresh variable $y$ that is added to the spine. We recursively expand both the freshly generated variable and the resulting spine, noting that these expansions happen at structurally smaller types. Our $\eta$-expansion assumes that the terms in the spine of a neutral are already $\eta$-long: it wraps the neutral in function abstractions, but does not traverse into it.

### 3.3.4 Hereditary Substitution

Hereditary substitution (Fig. 3.6) defines substitution in a way that preserves normal forms, and is used to compute the return types of dependent function applications. At first glance, many of the rules look like a traditional substitution definition. They traverse the expression looking for variables, and replace them with the corresponding term. However, the difference is in how the actual replacement of variables is handled.

Though the majority of the rules traverse terms looking for variables to replace, the key computation for hereditary substitution takes place in the rule SHsubRSpine. When replacing $x$ with $u_1$ in $xp\ u_2$, we find the substituted forms of $u_2$ and $xp$, which we call $u_3$ and $\lambda y.\ u'_1$ respectively. If the inputs are well typed and $\eta$-long, the substitution of the spine always returns a $\lambda$-term, meaning that its application to $u_3$ is not a normal form. To produce a normal form in such a case, we continue substituting, recursively replacing $y$ with $u_3$ in $u'_1$. A similar substitution in the codomain of $U$ gives our result type. Thus, if this process terminates, it always produces a normal form.

To ensure that the process does terminate for well typed inputs, we define hereditary substitution in terms of the type of the variable being replaced. Since we are replacing a different variable in the premise SHsubRSpine, we must track
the type of the resultant expression when substituting in spines, which is why substitution on neutral forms is a separate relation. Termination is shown by ordering types according to the multiset of universes of all arrow types that are subterms of the type, similar to techniques used for Predicative System F [54, 91]. We can use the well-founded multiset ordering given by Dershowitz and Manna [47]: if a type $U$ has maximum arrow type universe $\ell$, we say that it is greater than all other types containing fewer arrows at universe $\ell$ whose maximum is not greater than $\ell$. Predicativity ensures that, relative to this ordering, the return type of a function application is always less than the type of the function itself. Likewise, the order is preserved under substitution, because predicativity ensures the size of any terms replacing variables is dominated by the size of the type of the replaced variable. In all premises but the last two of SHSUBRSpine, we recursively invoke substitution on strict subterms, while keeping the type of the variable the same. In the remaining cases, we perform substitution at a type that is smaller by
**Figure 3.5:** SDTL: Auxiliary Judgments for Normal Forms
Figure 3.6: SDTL: Hereditary Substitution
our multiset order.

### 3.3.5 Properties of SDTL

Since SDTL is mostly standard, it enjoys the standard properties of dependently typed languages. Subject reduction is easily proved, and the termination of hereditary substitution means that the language is strongly normalizing, and hence logically consistent. Since the type rules, hereditary substitution, and normalization are syntax-directed and terminating, type checking is decidable. Finally, because all well-typed terms have normal forms, SDTL is type safe.

### 3.4 GDTL: Abstracting the Static Language

We now present GDTL, a gradual counterpart to SDTL derived by extending the AGT methodology [67] to the setting of dependent types. The key idea behind AGT is that gradual type systems can be designed by first specifying the meaning of gradual types in terms of sets of static types. This meaning is given as a concretization function \( \gamma \) that maps a gradual type to the set of static types that it represents, and an abstraction function \( \alpha \) that recovers the most precise gradual type that represents a given set of static types. Formally, \( \gamma \) and \( \alpha \) form a Galois connection, an important concept from abstract interpretation (hence the name Abstracting Gradual Typing). This property lets the gradual guarantee be easily proven. We explain more in Section 3.4.4.

Once the meaning of gradual types is specified using \( \gamma \) and \( \alpha \), the typing rules and dynamic semantics for the gradual language can be derived systematically. First, \( \gamma \) and \( \alpha \) let us lift the type predicates and type functions used in the static type system (such as equality, subtyping, join, etc.) to obtain their gradual counterparts. From these definitions, algorithmic characterizations can then be validated and implemented. Second, the gradual type system is obtained from the static type system by using these lifted type predicates and functions. Finally, the run-time semantics are defined by proof reduction of the typing derivation, mirroring the type safety argument at run time.

In GDTL, typing derivations are augmented with pieces of evidence for consis-
tent judgments, whose combination during reduction may be undefined, hence resulting in a run-time error. This ascription is evidence for plausible equality (consistency) in the AGT sense (Section 2.1.3.3), but our use of bidirectional typing for normal forms means we need only ascribe neutral terms. Also, because the evidence is a dependent type, we can perform hereditary substitution on it like any other term. Normally evidence is not present for compile-time type checking, but here it is needed because dependent type checking normalizes types, and gradual typing means those normalizations can fail.

In this chapter we follow the AGT methodology, specifying \( \gamma \) and \( \alpha \), then describing how the typing rules are lifted to gradual types. We uncover several points for which the standard AGT approach lacks the flexibility to accommodate full-spectrum dependent types with \( ? \) as a term. We describe our extensions to (and deviations from) the AGT methodology, and how they let us fully support gradual dependent types.

Throughout this section, we refer to not-yet-defined gradual versions of hereditary substitution and normalization. We leave the detailed development of these notions to Section 3.5, as they are non-trivial if one wants to preserve both decidable type checking and the gradual guarantees. The dynamic semantics of \( \text{GDTL} \) are presented in Section 3.6, and its metatheory in Section 3.7.

### 3.4.1 Terms and Normal Forms

The syntax of the \( \text{GDTL} \) surface language (Fig. 3.7) is an extension of \( \text{SDTL} \)'s syntax. We use green, italic serif font for metavariables denoting gradual surface terms. In addition to constructs from \( \text{SDTL} \), \( \text{GDTL} \)'s syntax includes \( ? \), the unknown term/-type, which represents a type or term that is unknown to the programmer. By annotating a term with type \( ? \) or leaving a term as \( ? \), they can let their program type check with only partial typing information or an incomplete proof.

The syntax for normal forms in \( \text{GDTL} \) is similar to \( \text{SDTL} \), but has some technical subtleties, since we use normal forms to define abstraction, concretization, and hereditary substitution. As with terms, \( ? \) is now a valid normal form. Neutral normals are ascribed with \textit{evidence}, a type that is more precise than both the type of the neutral and the type at which it is used, which witnesses the consistency of
\[ t, T ::= \lambda x. t \mid t_1 \mid t_2 \mid x \mid \text{Type}_\ell \]

\[ | (x : T_1) \rightarrow T_2 | t :: T | ? \]  

(Gradual Terms)

\[
\begin{aligned}
\text{GNE} \ni u, U & ::= \ ? | \text{Type}_\ell | \lambda x. u \\
& | (N)N | (\text{Type}_\ell)N \\
& | (?)(x : U_1) \rightarrow^\omega U_2
\end{aligned}
\]

(Gradual Normal Forms)

\[
\begin{aligned}
\text{GNE} \ni N & ::= xp \\
p & ::= \cdot | p u
\end{aligned}
\]

(Gradual Neutral Forms)

\[ \text{Figure 3.7: gdtl: Terms and Normal Forms} \]

those two types. We formalize precision in Section 3.4.2, but for now, it suffices to think of a term being less precise than another if some number of ? occurrences in the first can be replaced to obtain the second. Since \( \eta \)-longness ensures that neutrals do not have function types, this evidence is either neutral, \( \text{Type}_\ell \), or the completely unknown \( ? \). Function types are annotated with a level \( \ell \). The type \( (x : u_1) \rightarrow u_2 \) is well-formed at type \( \text{Type}_\ell \). Additionally, arrows may be annotated with the special maximum level \( \omega \) where \( (x : u_1) \rightarrow^\omega u_2 \) is well-formed at type \( ? \), which is used in places where \( ? \) makes it impossible to determine a maximum level for a function type. These annotations are necessary to ensure the termination of hereditary substitution, but are inferred during normalization, and are never present in source programs. We often omit these annotations, as they clutter the presentation.

Normal forms do not contain ascriptions. Though statically typed languages use ascriptions only to guide type checking, the potential for dynamic type failure means that ascriptions have computational content in gradual typing. Notably, only variables or neutral applications can synthesize \( ? \) as a type, though any typed expression can be checked against \( ? \). Omitting ascriptions gives us a form of canonicity: the type of a normal form conveys what possible forms it can take. For example, we can layer ascriptions on top of terms, such as \((true :: ?) :: ? \rightarrow ?\), but the only normal forms with function types are lambdas and ?.

76
3.4.2 Concretization and Predicates

The main idea of AGT is that gradual types abstract sets of static types, and that each gradual type can be made concretely as a set of static types given by a concretization function \( \gamma \). For dependent types, we extend this to say that gradual terms represent sets of static terms. In a simply-typed language, a static type embedded in the gradual language concretizes to the singleton set containing itself. However, for terms, we wish to consider the entire \( \alpha \beta \eta \)-equivalence class of the static term. As with type checking, this process is facilitated by considering only normal forms. The \texttt{gdtl} concretization function \( \gamma : \text{GNF} \rightarrow \mathcal{P}(\text{SNF}) \setminus \emptyset \), defined in Fig. 3.8, defines a non-empty set of static normal forms for each gradual term. It recurs over sub-terms, with \( ? \) mapping to \( \text{SNF} \), the set of all static normal forms (as defined in Fig. 3.2). It is defined mutually with \( \gamma_N \), which maps the set of gradual neutrals to sets of static neutrals from \( \text{SNE} \).

Given the concretization, we can lift a predicate from the static system to the gradual system. A predicate holds for a gradual type if it holds for some type in its concretization. The gradual lifting of equality is \textit{consistency}, \( U \equiv U' \), which holds if and only if \( \gamma(U) \cap \gamma(U') \neq \emptyset \). Figure 3.8 gives an equivalent syntactic definition. Concretization also gives us a notion of \textit{precision} on gradual types. We say that \( U \preceq U' \) if \( \gamma(U) \subseteq \gamma(U') \): that is, \( U \) is more precise because its concretization has fewer terms it could plausibly represent. We can similarly define the composition of two types \( U \cap U' \) as the most general term that is as precise as both \( U \) and \( U' \). In this chapter, \( U \cap U' \) is defined if and only if \( U \equiv U' \) i.e. \( \gamma(U) \cap \gamma(U') \neq \emptyset \), and like the consistency relation, it can be computed syntactically. In Chapter 5, we define a way of composing types that does not have this property, but better reflects the dynamic behaviour of terms.

3.4.3 Functions and Abstraction

Concretization is not enough to define all typing rules, since some rules match on the syntactic forms of their inputs. For example, when type checking a function application, we must handle the case where the function has type \( ? \). Recall the static rule \texttt{SSYNTTHAPP}, which had a premise that the applied function synthesized an arrow type. Since \( ? \) is not an arrow type, the static rule would fail in all such
\[ \gamma : \text{GNF} \to \mathcal{P}(\text{SNF}) \setminus \emptyset \]
\[ \gamma_N : \text{GNE} \to \mathcal{P}(\text{SNE}) \setminus \emptyset \]

\[
\begin{align*}
\gamma(?) & = \text{SNF} \\
\gamma(\text{Type}_r) & = \{ \text{Type}_r \} \\
\gamma(\lambda x. u) & = \{ \lambda x. u' | u' \in \gamma(u) \} \\
\gamma((x : U_1) \to U_2) & = \{ (x : U_1') \to U_2' | U_1' \in \gamma(U_1), U_2' \in \gamma(U_2) \} \\
\gamma((\text{Type}_r) N) & = \{ (\text{Type}_r) N | N \in \gamma_N(N) \} \\
\gamma(\text{Type}_r) N & = \{ (\text{Type}_r) N | u \in \text{Type}_r | t \in \mathbb{N} \} \cup \text{SNE}, N \in \gamma_N(N) \\
\gamma((N')N) & = \{ (N')N | N' \in \gamma_N(N'), N \in \gamma_N(N) \} \\
\gamma_N(x) & = \{ x \} \\
\gamma_N(x \ pu) & = \{ x \ pu' | x \ pu' \in \gamma_N(x p), u' \in \gamma(u) \}
\end{align*}
\]

\[
\begin{array}{ccc}
\text{ConsistentEq} & \text{ConsistentPi} & \text{ConsistentLam} \\
\hline
u \equiv u & U_1 \equiv U_1' & U_2 \equiv U_2' \\
\hline
\end{array}
\]

\[
\begin{array}{ccc}
\text{ConsistentDynL} & \text{ConsistentDynR} & \text{ConsistentNeuL} \\
\hline
? \equiv u & u \equiv ? & \langle ? \rangle N_1 \equiv \langle U \rangle N_2 \\
\hline
\end{array}
\]

\[
\begin{array}{ccc}
\text{ConsistentNeuL} & \text{ConsistentNeuR} & \text{ConsistentVar} \\
\hline
\langle ? \rangle N_1 \equiv \langle U \rangle N_2 & \langle ? \rangle N_1 \equiv \langle U \rangle N_2 & x \equiv x \\
\hline
\end{array}
\]

\[
\begin{array}{ccc}
\text{ConsistentApp} \\
\hline
x \ p \equiv x \ p' & u \equiv u' \\
\hline
x \ pu \equiv x \ p' u'
\end{array}
\]

Figure 3.8: GDTL: Concretization and Consistency
cases. Instead, we extract the domain and codomain from the type using partial functions. Statically, \( \text{dom} (x : U) \rightarrow U' = U \), and is undefined otherwise. But what should the domain of \(?\) be?

\(\text{AGT}\) gives a recipe for lifting such partial functions to gradual types using the counterpart to concretization: abstraction. The abstraction function \(\alpha\) is defined in Fig. 3.9. It takes a set of static terms, and finds the most precise gradual term that is consistent with the entire set.

With concretization and abstraction, we are able to take gradual terms to sets of static terms, perform operations on those sets, then transform the result set back into a gradual term. We can then define gradual partial functions in terms of their static counterparts: we concretize the inputs, apply the static function element-wise on all values of the concretization for which the function is defined, then abstract the output to obtain a gradual term as a result.

We use concretization and abstraction to define partial functions to decompose types and terms. For example, the domain of a gradual term \(U\) is:

\[ \alpha(\{\text{dom} U' \mid U' \in \gamma(U)\}) \]

which can be expressed algorithmically using the rules in Fig. 3.9. We define function-type codomains and lambda-term bodies similarly, though we pair these operations with substitution because after extracting the codomain from a function type or the body of a lambda, we typically want to substitute some term in, for example, when checking or normalizing an application. Thus, we avoid creating a "dummy" bound variable name for \(?\).

### 3.4.4 Soundness and Optimality

Taken together, \(\alpha\) and \(\gamma\) form a Galois connection, meaning it is sound and optimal. Soundness ensures that \(\alpha\) does not spuriously add information: for any set \(S\), \(\alpha S\) represents at least as many types as \(S\). Optimality expresses that \(\alpha\) does not lose precision unnecessarily: of the gradual types available, it chooses the most precise one that denotes the given set.

**Theorem 3.4.1** (Soundness). For all \(\emptyset \not\subseteq S \subseteq \text{SNF}, S \subseteq \gamma(\alpha(S))\).
\(\alpha : \mathcal{P}(\text{SNF}) \setminus \emptyset \rightarrow \text{GNF}\)
\(\alpha_N : \mathcal{P}(\text{SNE}) \setminus \emptyset \rightarrow \text{GNE}\)

\[
\begin{align*}
\alpha((x : U_1) \rightarrow U_2 \mid U_1 \in A, U_2 \in B)) &= (x : U'_1) \rightarrow U'_2 \quad \text{where } \alpha(A) = U'_1, \alpha(B) = U'_2 \\
\alpha(\lambda x. u \mid u \in A)) &= \lambda x. u' \quad \text{where } \alpha(A) = u' \\
\alpha(\{\text{Type}_i\}) &= \text{Type}_i \\
\alpha(\{yp'\}xp \mid p' \in \{A\}, p \in \{B\}) &= \{yp'\}xp \quad \text{where } \alpha_N(\{yp' \mid p' \in A\}) = yp', \\
\alpha_N(\{xp \mid p \in B\}) &= xp \\
\alpha(\{\text{Type}_i\}xp \mid p \in \{A\}) &= \{\text{Type}_i\}xp \quad \text{where } \alpha_N(\{xp \mid p \in A\}) = xp \\
\alpha(\{u\}xp \mid p \in \{A\}) &= \{?\}\alpha_N(S) \quad \text{otherwise} \\
\alpha_N(\{xp u \mid xp \in A, u \in B\}) &= x \ r' u' \quad \text{where } \alpha(A) = x \ p', \alpha(B) = u'
\end{align*}
\]

\textbf{Domain:} \(\text{GNF} \rightarrow \text{GNF} \) \textit{(Gradual Domain)}

\textbf{DomainP1:} \(\text{dom} \ (x : U_1) \rightarrow U_2 = U_1\)

\textbf{DomainDyn:} \(\text{dom} \ ? = ? \quad \text{undefined otherwise}\)

\textbf{CodSubP1:} \([u/x]^{U_1}U_2 = U'_2\)
\[\text{CodSubDyn} \quad [u/x]^{U_1}\text{Cod} = [u/x]\]

\textbf{CodSubDyn:} \([u/\_]\text{Cod} \ (x : U_1) \rightarrow U_2 = U'_2\)

\[\text{CodSubDyn} \quad [u/\_]\text{Cod} \ ? = ? \quad \text{undefined otherwise}\]

\textbf{Gradual Function Body Substitution}

\textbf{BodySubP1:} \([u/x]^{U}u_2 = u'_2\)
\[\text{BodySubDyn} \quad [u/\_]^{U}\text{Body} = ? \quad \text{undefined otherwise}\]

\[\text{BodySubDyn} \quad [u/\_]^{U}\text{Body} ? = ? \quad \text{undefined otherwise}\]

\textbf{Figure 3.9: GDTL: Abstraction and Lifted Functions}
See proof in Appendix A.

**Theorem 3.4.2** (Optimality). For \( \emptyset \not\subseteq S \subseteq \text{SNF} \) and \( U \in \text{GNF} \), if \( S \subseteq \gamma(U) \) then \( \alpha(S) \subseteq U \). That is, \( \gamma(\alpha(S)) \subseteq \gamma(U) \).

See proof in Appendix A.

### 3.4.5 Typing Rules

Given concretization and abstraction, AGT gives a recipe to convert a static type system into a gradual one, and we follow it closely. Figure 3.10 gives the rules for typing. Equalities implied by repeated metavariables have been replaced by consistency checks, such as in GCHECKSYNTH. Similarly, in GCHECKPI we use the judgment \( U \equiv \text{Type} \) to ensure that the given type is consistent to, rather than equal to, some \( \text{Type}_\ell \) when \( \ell \) is unknown. Rules that matched on the form of a synthesized type instead use partial functions, as we can see in GSYNTHAPP.

We split the checking of functions into GCHECKLAMPI and GCHECKLAMDYN for clarity, but the rules are equivalent to a single rule using a partial function.

The last thing we must decide is how to type the unknown term \( ? \). We wish to let \( ? \) replace any term in a program. But what should its type be? By the AGT philosophy, \( ? \) represents all terms, so it should synthesize the abstraction of all inhabited types, which is \( ? \). We encode this in the rule GSYNTHDYN. So we can use the unknown term in any context.

The consistency relation used in the GDTL typing rules corresponds to the gradual lifting of *definition equality*: \( u_1 \equiv u_2 \) if and only if some \( u'_1 \in \gamma(u_1) \) and \( u'_2 \in \gamma(u_2) \) where \( u'_1 =_{\alpha\beta\eta} u'_2 \). Because \( \gamma(x) = \{x\} \) for any variable \( x \), consistency, precision and meet are all well-defined on open terms, and two variables are consistent if and only if they are equal. This correspondence reflects our intensional approach: functions are compared according to their bodies, rather than their extensional behaviour. Chapter 5 explains the problems this intensional approach raises for run-time checks, but it is sufficient for the foundational calculus of this chapter. Though \( \gamma \) and \( \alpha \) are crucial for deriving the definitions of gradual operations, the operations can be implemented algorithmically as syntactic checks: an implementation does not need to compute \( \gamma \) or \( \alpha \).
As with the static system, we represent types in normal form, which makes consistency checking easy. Well-formedness rules (Fig. 3.11) are derived from the static system in the same way as the gradual type rules. The changes are GWFUNK, which lets ? check against any type as a normal form, and GWFNEUT replacing SWFNEUT and SWFNEUTTYPE, which lets a neutral be used as a normal form so long as it is tagged with valid evidence that its type is consistent with the target type. This evidence is some normal-form type that is as precise as the two consistent types. We also allow evidence to be neutral: neutrals are embedded into normal forms by tagging them with evidence, so allowing non-normal neutral evidence means that we avoid stacks of evidence-within-evidence. Evidence tracking is necessary for safe normalization of gradual terms. We explain compile-time evidence more, along with the full gradual normalization judgments $\Gamma \vdash t \Rightarrow U$ and $\Gamma \vdash u \leftarrow t \Leftarrow U$, in Section 3.5.3.
\[ \varepsilon ::= U \mid N \]  
*(Compile-time evidence)*

\[
\Gamma \vdash u \Rightarrow U \quad (Gradual Well Formed Neutrals)
\]

\[
\begin{align*}
\text{GWVFAR} & \quad \Gamma \vdash \text{xp} \Rightarrow (y : U_1) \rightarrow U_2 \\
(x : U) & \in \Gamma \\
\ downsquare \quad \Gamma \vdash x & \Rightarrow U \\
\downarrow \quad \Gamma & \vdash \text{xp} u \Rightarrow U_3
\end{align*}
\]

\[
\begin{align*}
\text{GWFNeut} & \quad \Gamma \vdash N \Rightarrow U' \quad \varepsilon \vdash U \equiv U' \\
\downarrow \quad \Gamma & \vdash (\varepsilon) N \Leftarrow U \\
\downarrow \quad \Gamma & \vdash \lambda x. u \Leftarrow (x : U_1) \rightarrow U_2
\end{align*}
\]

\[
\begin{align*}
\text{GWFUnk} & \quad \Gamma \vdash U : \text{Type} \\
\downarrow \quad \Gamma & \vdash ? \Leftarrow U \\
\downarrow \quad \Gamma & \vdash \varepsilon \equiv U
\end{align*}
\]

\[
\begin{align*}
\text{GWFLamUnk} & \quad \Gamma \vdash U_1 \Leftarrow \text{Type}_t \\
\downarrow \quad \Gamma (x : U_1) & \vdash U_2 \Leftarrow \text{Type}_t \\
\downarrow \quad \Gamma & \vdash \lambda x. u \Leftarrow (x : U_1) \rightarrow U_2
\end{align*}
\]

\[
\begin{align*}
\text{GWFPi} & \quad \Gamma \vdash U_1 \Leftarrow \text{Type}_t \\
\downarrow \quad \Gamma (x : U_1) & \vdash U_2 \Leftarrow \text{Type}_t \\
\downarrow \quad \Gamma & \vdash (x : U_1) \rightarrow U_2 \Leftarrow ?
\end{align*}
\]

\[
\begin{align*}
\text{GWFType} & \quad \Gamma \vdash \text{Type}_t \Leftarrow \text{Type}_{t+1} \\
\downarrow \quad \Gamma & \vdash \text{Type}_t \Leftarrow ?
\end{align*}
\]

\[
\begin{align*}
\text{GEvNorm} & \quad \Gamma \vdash U'' \Leftarrow U \quad U'' \Leftarrow U' \\
\downarrow \quad \Gamma & \vdash U'' \equiv U'
\end{align*}
\]

\[
\begin{align*}
\text{GEvNeutR} & \quad \Gamma \vdash N'' \Leftarrow N \quad N'' \Leftarrow N' \\
\downarrow \quad \Gamma & \vdash N'' \equiv \langle \text{Type}_t \rangle N
\end{align*}
\]

\[
\begin{align*}
\text{GEvNeutL} & \quad \Gamma \vdash N'' \Leftarrow \langle \text{Type}_t \rangle N \equiv ? \\
\downarrow \quad \Gamma & \vdash N'' \equiv \langle \text{Type}_t \rangle N
\end{align*}
\]

\[
\begin{align*}
\text{GEvNeutUnk} & \quad \Gamma \vdash N'' \Leftarrow ? \equiv \langle \text{Type}_t \rangle N \\
\downarrow \quad \Gamma & \vdash N'' \equiv ?
\end{align*}
\]

*Figure 3.11: GDTL: Well-Formedness Rules*
3.4.6 Example: Type checking the head of an empty vector

To illustrate how the GDTL type system works, we explain the type checking of one example from the introduction. Suppose we have postulated types for natural numbers and vectors, and a derivation for:

\[ \Gamma \vdash \text{head} \Rightarrow (A : \text{Type}_0) \rightarrow (n : \text{Nat}) \rightarrow \text{Vec} A (n + 1) \rightarrow A. \]

In Fig. 3.12, we show the (partial) derivation of:

\[ \Gamma \vdash \text{head Nat 0 (Nil :: Vec Nat ?)} \Rightarrow \text{Nat} \]

The key detail here is that the compile-time consistency check lets us compare 0 to ?, and then ? to 1, which lets the example type check. Notice how we only check consistency when we switch from checking to synthesis. Though this code type checks, it fails at run time. Once we have defined semantics for GDTL, we explain the execution of this term in Section 3.6.4.
Figure 3.12: Type Derivation for head of nil
### 3.5 Approximate Normalization

Not all type checking computations are simple substitutions like we saw in the previous example. As we saw in Section 3.2.3, the type-term overlap in GDTL means that code that is run during type checking may fail or diverge. But what is the meaning of such errors inside of a type?

One option is to declare types ill-formed if they contain imprecisely typed terms. However, this approach breaks the criteria for a gradually typed language, since dependent functions could never take imprecise terms as arguments. In particular, it would result in a language that violates the static gradual guarantee (Section 3.2.4). The static guarantee implies that if a program does not type check, the programmer knows that the problem is not the absence of type precision, but that the types present are fundamentally wrong. Increasing precision in multiple places never causes a program to type check if doing so in one place fails.

This section gives the definitions of normalization and substitution that were missing from the previous chapter. We describe approximate hereditary substitution, which regains decidability while preserving the gradual guarantee, by producing compile-time normal forms that are potentially less precise than their runtime counterparts. Thus, we trade precision for a termination guarantee.

Approximate substitution is used to build approximate normalization. In addition to avoiding non-termination, approximate normalization handles dynamic type failures by tracking type information of neutrals. A key insight of this chapter is that we need separate notions of compile-time normalization and run-time execution. That is, we use approximate normalization only in our types. Executing our programs at run time does not lose information, but may diverge or fail.

For type checking, the effect of approximate normalization is that non-equal terms of the unknown type may be indistinguishable at compile time. Returning to the example from Section 3.2.3, the user’s faulty factorial-length vector type checks, but at type $\text{Vec Nat} \, ?$. Using it never raises a static error due to its length, but it may raise a run-time error. The $\text{failList}$ example from Section 3.2.3 raises a type error, but $\text{failFun}$ only raises a type error when called on an invalid input.
3.5.1 Differences from the Published Version

In this section, we present a novel notion of compile-time evidence for GDTL. The previous version of this work [58] erroneously claimed that its version of GDTL satisfied the static gradual guarantee, but previous approach of normalizing errors to ? was flawed. We develop compile-time evidence as a way to handle errors without producing ? as the result of dynamic errors in types.

The published version of this work contained a notion that this chapter lacks: ideal substitution, which was defined in terms of equivalence classes of the concretization of gradual terms. Ideal substitution provided theoretical justification for the definition of approximate substitution. Because that version approximated type errors to ?, the ideal normal form of a term was always as precise as the approximate normal form. However, error approximation meant that said version of GDTL did not satisfy the static gradual guarantee, a mistake not caught until after publication.

To handle errors in this chapter, we use a system of compile-time evidence that ensures we satisfy the static guarantee. Such a version does not approximate ideal substitution, because there are cases where a term could execute safely but raises an error. For example, $\lambda(x : Type). x$ can be safely applied to any term, despite the type ascription, so its $\alpha\beta\eta$-equivalence class is unrestricted. However, its approximate normalization ascribes the argument with evidence that prevents it from being in contexts incompatible with $Type$. We feel the loss of the relationship to ideal substitution is justified by gaining the static gradual guarantees.

3.5.2 Approximate Substitution

We now turn to the problem of how to recover decidable comparison for normal forms without losing the gradual guarantees. We again turn to (gradual) normal forms as representatives of $\alpha\beta\eta$-equivalence classes. What happens when we try to construct a hereditary substitution function syntactically, as in SDTL?

3.5.2.1 Ensuring Termination

The issue with gradualizing hereditary substitution is in adapting the rule SHS\textsc{UBR-Spine}. Suppose we are substituting $u$ for $x$ in $xp\ u_2$, and the result of substituting
in $xp$ is $\lambda y. u' : ?$. Following the $\texttt{agt}$ approach, we can use the $\texttt{dom}$ function to
calculate the domain of $?$, which is the type at which we substitute $y$. But such
a substitution violates the well-foundedness condition we imposed in the static
case! The type $?$ contains zero arrow-types, so a recursive substitution that is also
at type $?$ is not strictly smaller. Since the domain of $?$ is $?$, eliminating redexes
may infinitely apply substitutions without decreasing the size of the type. The
remaining cases have no termination issues, since the term we are substituting
into is structurally decreasing.

Hereditary substitution informs us as to the cases we must approximate to
preserve decidability. To guarantee termination, we must not perform recursive
substitutions in spines with $?$ typed heads. We are left with two options for how
to proceed without making recursive calls: we either fail when we try to apply
a $?$ typed function, or we return $?$. The former preserves termination, but it does
not preserve the static gradual guarantee. Reducing the precision of a well typed
program’s ascriptions should never yield ill typed code. If applying a dynami-
cally typed function caused failure, then changing an ascription to $?$ could cause
a previously successful program to be ill-typed, violating the guarantee.

Our solution is to produce $?$ when applying a function of type $?$. We highlight
the changes to neutral hereditary substitution in Fig. 3.13. GHsubRDynType ac-
counts for $?$ typed functions, and GHsubRDynSpine accounts for $?$ applied as a
function.

We must give gradual hereditary substitution one more check to guarantee
termination, because $? : ?$ could be used to circumvent the universe hierarchy.
For instance, we can assign $(x : ?) \to (x \ Type_0)$ the type $Type_0$, and we can even
write a version of Girard’s paradox [38, 68] by using $?$ in place of $Type$.

To avoid Girard’s paradox, normal form arrow types are ascribed with a level,
and GHsubRlamSpine checks whether this value is decreasing according to our
metric. We extend the natural number order with a maximum element $\omega$, where
$i < \omega$ for every $i$. The level $\omega$ denotes a function type of unknown level. Then
$U < U'$ when the multiset of annotations on arrow types in $U$ is less than that of
$U'$ by the well-founded multiset ordering given by Dershowitz and Manna [47].
In the static case, the type of substitution is always decreasing for this metric.
In the presence of $?$, we must check if the order is violated and return $?$ if it is,
\[
[u/x]^U x \ p = u_2 : U_2
\]  
(Approximate Neutral Hereditary Substitution, Key Rules)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| GHSUBRHEAD | \[
[u/x]^U x \cdot = u : U \\
[u/x]^U x \ p = ? : (y : U_1) \rightarrow U_2 \\
[u/x]^U x \ p = u_2 : U_2 \\
[u/x]^U x \ p = [u_2/y]^U_1 U_2 = U_3 \\
\] |
| GHSUBRDYNSpINE | \[
[u/x]^U x \ p = ? : (y : U_1) \rightarrow U_2 \\
[u_2/y]^U_1 U_2 = U_3 \\
\] |
| GHSUBRDYNType | \[
[u/x]^U x \ p = u_2 : ? \\
[u_2/y]^U_1 U_2 = U_3 \\
\] |
| GHSUBRLAMSpINE | \[
[u/x]^U x \ p = \lambda y. u_2 : (y : U_1) \rightarrow U_2 \\
[u/x]^U x \ p = u_3 : U_1 < U \\
[u/x]^U x \ p = [u_3/y]^U_1 U_2 = U_3 \\
\] |
| GHSUBRLAMSpINEOrd | \[
[u/x]^U x \ p = \lambda y. u_2 : (y : U_1) \rightarrow U_2 \\
[u/x]^U x \ p = u_3 : U_1 < U \\
\] |

Figure 3.13: Approximate Neutral Substitution, Key Rules

as seen in the rule GHSUBRLAMSpINEOrd. Unlike applying a function of type ?, we believe that this case is unlikely to arise in practice unless programmers are deliberately using ? to circumvent the universe hierarchy.

### 3.5.2.2 Handling Type Imprecision

On top of substituting in neutral forms, hereditary substitution must handle term and type imprecision in normal forms, which we do in Fig. 3.14. For term imprecision, substituting in ? produces ? (GHSUBUNK). For type imprecision, because of the normal typing rule GWFCHECKSYNTH, hereditary substitution must account for how the actual type of an evidence-ascribed neutral term may differ from the type at which it is used. Moreover, the result should incorporate a substituted version of the type information contained within the evidence. These rules end up being a mess of cases, which is part of why we ultimately abandon hereditary substitution in later chapters.

If a neutral term is ascribed neutral evidence, then the evidence head may be the variable we are replacing. Rule GHSUBNEUTTySAME handles this case by finding the substituted form of the evidence, substituting in the spine of the term, then \(\eta\)-expanding the term at the evidence type (Fig. 3.15). Since \(\eta\)-expansion takes neutrals to normals, the result is ascribed evidence based on the given type.
\[
\begin{align*}
[u/x]^U u_1 &= u_2 & \text{(Approximate Normal Hereditary Substitution, Key Rules)} \\
\text{GSubNeutTySAME} \
x \neq y & [u/x]^U x p_1 = U_2 : U_3 \\
U_3 \cong \text{Type} & [u/x]^U p_1 = p_2 \\
y p_2 \sim y u_2 & u_2 : U_2 \\
\text{GSubNeutTyDIFF} \
x \neq y & x \neq z \\
[u/x]^U p_1 = p_2' & [u/x]^U p_1 = p_2 \\
\text{GSubUnk} & [u/x]^U (xp_1') yp_1 = u_2 \\
\text{GSubNeutType} \
x \neq y & [u/x]^U p_1 = p_2 \\
[u/x]^U \langle \text{Type}_\ell \rangle yp_1 = \langle \text{Type}_\ell \rangle yp_2 \\
\text{GSubUnk} & [u/x]^U \langle ? \rangle yp_1 = \langle ? \rangle yp_2 \\
\text{GSubSpineUnk} \
[u_1/x]^U xp = u_2 : U'' & [u_1/x]^U xp = u_2 \\
[u_1/x]^U \langle ? \rangle xp = u_2 & [u_1/x]^U \langle ? \rangle xp = \langle ? \rangle yp_2 \\
\text{GSubSpineTypeUnk} \
[u_1/x]^U xp = ? : U'' & [u_1/x]^U xp = \langle \text{Type}_\ell \rangle xp = \langle \text{Type}_\ell \rangle xp \\
[u_1/x]^U \langle \text{Type}_\ell \rangle xp = ? & [u_1/x]^U \langle \text{Type}_\ell \rangle xp = \langle \text{Type}_\ell \rangle xp \\
\text{GSubSpineTypeUnk} \
[u_1/x]^U xp = \langle \epsilon \rangle N : U'' & [u_1/x]^U xp = \langle \epsilon \rangle N \quad U'' \cong \text{Type}_\ell \\
[u_1/x]^U \langle \text{Type}_\ell \rangle xp = \langle \epsilon \rangle N & [u_1/x]^U \langle \text{Type}_\ell \rangle xp = \langle \epsilon \rangle N \quad U'' \cong \text{Type}_\ell \\
\text{GSubSpineNeutUnk} \
[u_1/x]^U xp' = p_2' & [u_1/x]^U xp_1 = \langle \epsilon \rangle z p_2 : U_2 \\
\text{GSubSpineNeutUnk} \
[u_1/x]^U (yp_1') xp_1 = \langle yp_1' \cap \epsilon \rangle z p_2 & [u_1/x]^U (yp_1') xp_1 = \langle yp_1' \cap \epsilon \rangle z p_2 \\
\text{Type} = & (x : U_1) \rightarrow U_2 : U'' \\
\text{GSubSpineTypeP1} \
[u_1/x]^U xp = \langle \epsilon \rangle N & [u_1/x]^U xp = \langle \epsilon \rangle N \quad U'' \cong \text{Type}_\ell \\
\text{GSubSpineNeutNeut} \
[u_1/x]^U xp = \langle \epsilon \rangle N & [u_1/x]^U xp = \langle \epsilon \rangle N \quad U'' \cong \text{Type}_\ell \\
\end{align*}
\]

**Figure 3.14**: GDTL: Approximate Normal Substitution
By expanding at the type given by the substituted evidence, its type information is retained in the ascriptions on the result, while still accounting for the fact that the new type may be a function type. If the head is a different variable, we substitute in both spines (GHsubNeutTyDIFF), and if the evidence is not neutral, then we substitute in the spine of the neutral term (GHsubNeutType,GHsubNeutUnk).

The final case to handle is when replacing $x$ in an evidence-ascribed neutral whose head is $x$. Since the result may not be neutral, we cannot simply tack on a substituted version of the old evidence. However, we take advantage of an important fact about the typing of normals in GDTL: only neutral terms and $\forall$ can have neutral types. We handle the cases separately:

- If a neutral has evidence $\forall$, then the evidence carries no information, and we can disregard it (GHsubSpineUnk).

- If a neutral has evidence $Type_\ell$, then the result must be a type. If it is $\forall$, a universe, or a function type, then we need not propagate the evidence (GHsubSpineTypeType, GHsubSpineTypePi). If the result is neutral, then we use composition to ensure that the evidence on that neutral is consistent with $Type_\ell$ (GHsubSpineTypeNeut).

- If a neutral term has neutral evidence with head $y$, then it must synthesize either $\forall$ or another neutral with head $y$. Predicativity ensures that $y$ cannot be equal to $x$, so replacing $x$ in the evidence produces another $y$-headed neutral. If the result of substituting on the neutral term is $\forall$, then the final result is $\forall$, and we need not propagate evidence (GHsubSpineNeutUnk). Otherwise, substituting produces a non-$\forall$ term, whose type needs to be consistent with $y$-headed neutral evidence, meaning that the substitution should fail unless the result is neutral. When the result is neutral, we compute new $y$-headed evidence by substituting, and compose with the evidence on the result of substituting in the neutral term (GHsubSpineNeutNeut). So we produce a neutral that reflects the type information of the substituted initial evidence.
$N \leadsto_{\eta} u : U$ (Gradual $\eta$-Expansion)

<table>
<thead>
<tr>
<th>GE\textsc{taNeut}</th>
<th>GE\textsc{taType}</th>
<th>GE\textsc{taUnk}</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N \leadsto_{\eta} (N') N \bowtie (\text{Type}_i) N'$</td>
<td>$N \leadsto_{\eta} (\text{Type}_i) N : \text{Type}_i$</td>
<td>$N \leadsto_{\eta} (?) N : ?$</td>
</tr>
</tbody>
</table>

\[
\text{GE\textsc{tApi}} \quad \frac{y \leadsto_{\eta} u' : U_1 \quad x \leadsto_{\eta} u : U_2}{x \leadsto_{\eta} \lambda y. u : (y : U_1) \to U_2}
\]

**Figure 3.15:** gd\textsc{tl}: $\eta$-expansion

### 3.5.3 Normalization and Compile-Time Evidence

Approximate hereditary substitution eliminates non-termination, but we must still account for dynamic failures when adapting the normalization relation of sd\textsc{tl} (Fig. 3.4) to gd\textsc{tl}. We do this by augmenting neutral terms with evidence, and by introducing a partial casting meta-operation to convert a normal form from one type to another. We explain the theory behind evidence more in Section 3.6. The main idea is that, when a neutral synthesizes a type and is checked at a consistent type, that consistency implies that there exists some type as precise as both the synthesized and checked-against types. We attach one such type to a neutral term, to represent all the collected run-time type information about the neutral, and whenever that neutral is used at a different type, we compose the evidence with the new type, so that it monotonically evolves. This monotonic evolution is central to the simplicity of the ag\textsc{t} approach to proving the gradual guarantees.

The possibility for errors arises at the boundary between checking and synthesis, i.e. the normalization analogue of sc\textsc{heckSynth}. The critical property of normalization, around which we design the relation, is type preservation: if $\Gamma \vdash u \leftarrow t \Leftarrow U$, then we should have both $\Gamma \vdash t \Leftarrow U$ and $\Gamma \vdash u \Leftarrow U$. For the checking-synthesis boundary, if $\Gamma \vdash t \leadsto u' \Rightarrow U'$, and $\Gamma \vdash U' \equiv U$, then normalizing $t$ while checking against $U$ should produce a term $u$ such that $\Gamma \vdash u \Leftarrow U$. The difficulty is that normalization does not preserve type synthesis, only type check-
ing. For example, applications always synthesize, but if the applied expression is a λ-term, then the result could normalize to another λ-term, which does not synthesize a type. Likewise, t could be some ascribed term t′ :: T, where t′ synthesizes a type consistent with T but not with U, since consistency is not transitive. So u′ is not guaranteed to check against type U, since we cannot necessarily apply GWFCheckSynth.

Because of dynamic type errors during normalization, not all well typed terms are guaranteed to have a normal form. For GDTL, this design choice has the unfortunate effect that, when typing an application t₀ t₁, if t₁ contains a run-time type error then the application does not type check. To type t₀ t₁, the rule GSynthApp normalizes t₁ to substitute into the return type. For example, if tyld is defined as \( \lambda x. x : Type_0 \rightarrow Type_0 \), then tyld (Type₀ :: ? :: Type₀) is ill typed, even though Type₀ :: ? :: Type₀ is well typed, because it does not have a normal form.

By tagging neutrals with evidence, we are able to normalize terms with variables without necessarily raising an error, even if some values for those variables that would raise an error. When variables are replaced with concrete terms, the type information is checked against the type information in the term. For example, \( \lambda x. x :: Type_0 \rightarrow Type_0 \). tyld(x :: Type₀) is well typed, even though x :: Type₀ might fail for some inputs. The key is that x normalizes to (?x) at type ?. To normalize to a result of type Type₀, the evidence is updated to produce \( \langle Type_0 \rangle x \), which checks against Type₀ because it synthesizes the consistent type ?. So until a concrete value for x is given, no error is raised. The variable x has a normal form at type Type₀ that captures the known type information without needing to ensure that all possible x values match Type₀. If, for example, Type₉₉ is supplied for x, the hereditary substitution GHsubSpineType rule can not be applied because 0 ≠ 99 + 1. In later chapters, we can further defer errors by treating casts and composition as operations in the language, rather than meta-operations. Eremondi et al. [58] incorrectly claimed that allowing dynamic type errors during normalization would prevent t₀ t₁ from being well typed if t₁ encountered an error for any possible values of its free variables. However, the incarnation of GDTL presented in this dissertation does not suffer from this issue. For example, if t₁ is \( 1 + (true :: \mathbb{N}) \), then t₀ t₁ will not type check, but if t is \( 1 + x \), it does type check, even though there are values for x which would cause failure (namely true :: \mathbb{N}).
3.5.3.1 The Top Level Judgment: Approximate Normalization

The definition of approximate normalization is given in Fig. 3.16, defined in terms of casting and composition operations whose definition we leave for Section 3.5.3.2. The context and term are inputs, with the normal form as an output. The type is an output for synthesis rules, but an input for the checking rules. The rules primarily mirror the static rules, with a few major departures. Initial evidence for variables is generated during $\eta$-expansion (Fig. 3.15) in GNORMSYNTHVAR. GNORMCHECKSYNTH normalizes a term checking against $U$ by finding the synthesizing normal form for that term, and then casting the result to type $U$. The cast uses identical contexts to start, but functions or function types may cause these to differ during casting. Rule GNOCHECKPIUNK and CNOCHECKLAMDYN mirror GCHECKPIUNK and GCHECKLAMDYN: since functions and function types are checking forms, they cannot check against $\ ?$ with GNSYNTCHCHECK, so they need special rules to check against $\ ?$.

The possibility of $\ ?$ being applied as a function is handled in GNSYNTTHAPP: instead of matching on the normal form of the function and checking if it is a lambda, the body metafunction (Fig. 3.9), defined in terms of abstraction $\alpha$, produces $\ ?$ if the applied function is $\ ?$, and uses hereditary substitution to reduce the function application otherwise.

Normalization is also where we generate the annotations necessary to ensure the decreasing metric of hereditary substitution. As we see in the rules GNOCHECKPITYPE and GNOCHECKPIUNK, we annotate arrows either with the level against which they are checked, or with $\omega$ when checking against $\ ?$. The remaining rules for normalization directly mirror the rules from Fig. 3.10. Type, $\ ?$, and variables all normalize to themselves, and all other rules construct normal forms from the normal forms of their subterms.

3.5.3.2 Casting and Composition

Figures 3.17 to 3.19 define the key relation for handling type imprecision during normalization: a casting meta-operation $\Gamma_1|\Gamma_2 \vdash \langle U_2 \Leftarrow U_1 \rangle u_1 \Rightarrow u_2$, which takes as input two contexts $\Gamma_1$ and $\Gamma_2$, two types $U_1$ and $U_2$, and a term $u_1$ such that $\Gamma_1 \vdash u_1 \Leftarrow U_1$, and (if successful) outputs a term such that $\Gamma_2 \vdash u_2 \Leftarrow U_2$. The two
\[ \Gamma \vdash t \leadsto u \Rightarrow U \quad (\text{Approximate Normalization Synthesis}) \]

\[
\begin{align*}
\text{GNSynthAnn} & \quad \Gamma \vdash U \leftrightarrow T : \text{Type}_i \\
\Gamma \vdash u \leftrightarrow t \leftrightarrow U \\
\Gamma \vdash (t :: T) \leadsto u \Rightarrow U \\
\text{GNSynthApp} & \quad \Gamma \vdash t_1 \leadsto u_1 \Rightarrow U \\
\Gamma \vdash u_2 \leftrightarrow t_2 \leftrightarrow U_1 \\
[u_2/\cdot]_{T_i} & \text{body } u'_1 = u_3 \\
\Gamma \vdash t_1 \; t_2 \leadsto u'_3 \Rightarrow U_2 \\
\end{align*}
\]

\[
\begin{align*}
\text{GNSynthVar} & \quad (x : U) \in \Gamma \\
x \leadsto_{\eta} u : U \\
\Gamma \vdash x \leadsto u \Rightarrow U \\
\Gamma \vdash ? \leadsto ? \Rightarrow ? \\
\text{GNSynthDyn} & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash u \leftrightarrow t \leftrightarrow U \quad (\text{Approximate Normalization Checking}) \\
\text{GNCheckSynth} & \quad \Gamma \vdash t \leadsto u' \Rightarrow U' \\
U & \equiv U' \\
\Gamma | \Gamma \vdash (U \iff U') \; u' \leadsto u \\
\Gamma \vdash u \leftrightarrow t \leftrightarrow U \\
\text{GNCheckPiType} & \quad \Gamma \vdash U_1 \leftrightarrow T_1 \iff \text{Type}_i \\
\Gamma & \vdash (x : U_1) \Gamma \\
(x : U_1) \Gamma & \vdash U_2 \leftrightarrow T_2 \iff \text{Type}_i \\
\Gamma & \vdash (x : U_1) \Gamma \vdash (x : T_1) \rightarrow T_2 \iff \text{Type}_i \\
\end{align*}
\]

\[
\begin{align*}
\text{GNCheckPiUnk} & \quad \Gamma \vdash U_1 \leftrightarrow T_1 \iff ? \\
(x : U_1) \Gamma & \vdash U_2 \leftrightarrow T_2 \iff ? \\
\Gamma & \vdash (x : U_1) \Gamma \vdash (x : T_1) \rightarrow T_2 \iff ? \\
\end{align*}
\]

\[
\begin{align*}
\text{GNCheckLamPi} & \quad \Gamma \vdash x : U_1 \Gamma \vdash u \leftrightarrow t \leftrightarrow U_2 \\
\Gamma & \vdash \lambda x . u \leftrightarrow \lambda x . t \iff (x : U_1) \rightarrow U_2 \\
\text{GNCheckLamDyn} & \quad \Gamma \vdash x : ? \Gamma \vdash u \leftrightarrow t \iff ? \\
\Gamma & \vdash \lambda x . u \leftrightarrow \lambda x . t \iff ? \\
\end{align*}
\]

**Figure 3.16:** **GDTL:** Approximate Normalization
contexts are necessary because when a cast changes the domain of a dependent function type, it changes the context in which the result codomain is typed. We emphasize that casting is not a GDTL operation, but a relation between GDTL terms, which is in contrast to later chapters, in which we use cast calculi to define semantics for gradual languages. Though AGT is typically able to avoid casting by using evidence, the dependence of types on terms and the possibility of free variables in types means that we need a casting operation to ensure that evidence ascribed to terms is itself well typed.

To support the combination of different evidence, we use the composition operator (Fig. 3.20), which computes the precision meet of two terms. That is, it yields the most general term that is as precise as both inputs. Here composition is a metafunction, but in Chapter 5 we see a version of the meet that is built into the gradual language itself.

The main idea behind casting is to propagate type constraints deeper into terms until we hit a neutral term, at which point we can incorporate the new type into the evidence stored at that term. Neutrals are retyped with an auxiliary relation $\Gamma_1 \vdash (N_1 : U_1) \leadsto (N_2 : U_2)$, which takes a neutral that is typed under $\Gamma_1$ and casts terms in its spine to be well typed under $\Gamma_2$. Along with the produced neutral term, the types of both input and output neutrals are outputs, since they can be determined by synthesis. In MCASTNEUTToNEUT, this relation is used to create a well typed version of the neutral and its evidence under the new context. Finally, the result neutral is produced, with evidence formed by composing the (retyped) original evidence, the type that the result neutral synthesizes, and the type to which it is being cast. Composing with the retyped version of the old evidence makes it easy to establish that the monotonicity of casting follows from the monotonicity of composing. This in turn simplifies the proof of the static gradual guarantee. Most rules work similarly, or are simpler. For example, in rules MCASTNeuTFromUnkToUnk and MCASTNeuTEvTypeUnk, the evidence on the neutral is either $?$ or $\text{Type}_1$, avoiding the need for composition.

The remaining cases primarily propagate casts. MCASTUnk casts $?$ to itself at any type. The function type and function casting rules require having separate contexts for the input and output. In MCASTType, casting $\text{Type}_1$ to type $\text{Type}_{i+1}$ produces $\text{Type}_i$. The rule MCASTPi pushes a cast from $\text{Type}_i$ to $\text{Type}_i$ deeper.
\( \Gamma_1 \vdash \langle U_2 \leftrightarrow U_1 \rangle u_1 \rightsquigarrow u_2 \)  
(Meta-level Casting of Normal Forms)

\[
\text{MCastUnk} \\
\Gamma | \Gamma' \vdash \langle U_2 \leftrightarrow U_1 \rangle \rightsquigarrow ?
\]

\[
\text{MCastNeutToNeut} \\
\Gamma | \Gamma' \vdash \langle N_\text{val} : \langle \text{Type}_\ell \rangle \rangle \rightsquigarrow \langle N'_\text{val} : \langle \text{Type}_\ell \rangle \rangle \\
N_\text{ev} \vdash U \equiv \langle \text{Type}_\ell \rangle \rightsquigarrow \langle \text{Type}_\ell \rangle \\
\Gamma | \Gamma' \vdash \langle N_\text{ev} : U \rangle \rightsquigarrow \langle N'_\text{ev} : U' \rangle \\
\Gamma | \Gamma' \vdash \langle \langle \text{Type}_\ell \rangle \rangle N'_\text{chk} \equiv U \rangle \langle N_\text{ev} \rangle \rightsquigarrow \langle N'_\text{val} \cap N'_\text{ev} \cap N'_\text{chk} \cap N'_\text{val} \rangle 
\]

\[
\text{MCastNeutFromUnkToUnk} \\
\Gamma | \Gamma' \vdash \langle N' : ? \rangle \rightsquigarrow \langle N' : ? \rangle \\
\text{MCastNeutEvType} \\
\Gamma | \Gamma' \vdash \langle N' : U'_i \rangle \rightsquigarrow \langle N' : U' \rangle \\
U' \equiv \text{Type}_\ell \\
\Gamma | \Gamma' \vdash \langle U' \equiv U \rangle \langle \text{Type}_\ell \rangle N \rightsquigarrow \langle \text{Type}_\ell \rangle N' \\
\text{MCastType} \\
\Gamma | \Gamma' \vdash \langle \text{Type}_{\ell+1} \equiv \text{Type}_{\ell+1} \rangle \text{Type}_\ell \rightsquigarrow \text{Type}_\ell \\
\text{MCastPi} \\
\Gamma | \Gamma' \vdash \langle \text{Type}_\ell \equiv \text{Type}_\ell \rangle U_1 \rightsquigarrow U'_1 \\
\Gamma | \Gamma' \vdash \langle x : U \rangle \langle x : U'_i \rangle \Gamma' \vdash \langle \text{Type}_\ell \equiv \text{Type}_\ell \rangle U_2 \rightsquigarrow U'_2 \\
\Gamma | \Gamma' \vdash \langle \text{Type}_\ell \equiv \text{Type}_\ell \rangle (x : U_1) \rightarrow U_2 \rightsquigarrow (x : U'_1) \rightarrow U'_2 \\
\text{MCastLam} \\
\langle x : U \rangle \Gamma | \Gamma' \vdash \langle x : U'_i \rangle \Gamma' \vdash \langle U'_2 \equiv U_2 \rangle \rightarrow \langle \lambda x. \rightarrow u' \rangle \\
\Gamma | \Gamma' \vdash \langle x : U'_2 \rangle \rightarrow \langle x : U \rangle \rightarrow U_2 \lambda x. \rightarrow u \rightsquigarrow \lambda x. \rightarrow u' 
\]

\( \Gamma | \Gamma' \vdash \langle N : U \rangle \rightsquigarrow \langle N' : U' \rangle \)  
(Neutral Re-typing)

\[
\text{NeReTypeVar} \\
\langle x : U \rangle \in \Gamma \Rightarrow \langle x : U' \rangle \in \Gamma' \\
\Gamma | \Gamma' \vdash \langle x : U \rangle \rightsquigarrow \langle x : U' \rangle 
\]

\[
\text{NeReTypeApp} \\
\Gamma | \Gamma' \vdash \langle x p : U \rangle \rightsquigarrow \langle x p' : U' \rangle \\
\langle \text{dom} U' \equiv \text{dom} U \rangle \rightarrow \langle \text{dom} U' \equiv \text{dom} U \rangle \\
\Gamma | \Gamma' \vdash \langle x p u : U_2 \rangle \rightsquigarrow \langle x p' u' : U'_2 \rangle \\
\Gamma | \Gamma' \vdash \langle x p u : U_2 \rangle \rightsquigarrow \langle x p' u' : U'_2 \rangle 
\]

Figure 3.17: GDTL: Meta-level Casting
Figure 3.18: gdTL: Meta-level Casting to the unknown type

into the type, which is necessary because the result may need to be typed in a
different context from the input. Likewise, because function types bind a variable
\( x \), the result of casting the codomain must be typed in a context where the type of
\( x \) is the result of casting the domain. Similarly, in MCASTLAM functions are cast
by casting their bodies, but the resulting body must be typed with the bound \( x \)
having the result domain type.

Each type former has a rule for how to cast from a type using that former to
\( \) (Fig. 3.18). Functions, function types, and universes are cast by pushing casts
deepener into the term (MCASTLAMToUnk, MCASTPiToUnk, MCASTTypeToUnk).
\[ \Gamma_1 \vdash (U_2 \triangleq U_1) u_1 \leadsto u_2 \]  
(Meta-level Casting of Normal Forms, ctd.)

\[ \text{MCastTypeFromUnk} \]
\[
\Gamma \vdash (Type_{\ell+1} \triangleq ?) Type_{\ell} \leadsto Type_{\ell}
\]

\[ \text{MCastPiFromUnk} \]
\[
\Gamma \vdash (Type_{\ell} \triangleq ?) U_1 \leadsto U_1' \quad (x : ?) \Gamma \vdash (Type_{\ell} \triangleq ?) U_2 \leadsto U_2'
\]
\[
\Gamma \vdash (Type_{\ell} \triangleq ?) (x : U_1) \leadsto U_2 \leadsto (x : U_1') \leadsto U_2'
\]

\[ \text{MCastNeutTagUnkFromUnkToType} \]
\[
\Gamma \vdash (Type_{\ell} \triangleq ?) (N : ?) \leadsto (N' : U') \quad U' \equiv Type_{\ell}
\]

\[ \text{MCastLamFromUnk} \]
\[
(x : ?) \Gamma \vdash (U_2 \triangleq ?) u \leadsto u'
\]
\[
\Gamma \vdash ((x : U_1) \leadsto U_2 \triangleq ?) \lambda x. u \leadsto \lambda x. u'
\]

\[ \text{MCastNeutFromUnkToPi} \]
\[
\Gamma \vdash (N : ?) \leadsto (\eta : U') \quad x \leadsto_u u_1 \quad (x : U_1) \leadsto U_2 \equiv U'
\]
\[
\Gamma \vdash ((x : U_1) \leadsto U_2 \triangleq ?) (\eta : \eta) \leadsto \lambda x. y p u
\]

**Figure 3.19:** gdTL: Meta-level Casting from the unknown type

\[ \vdash (\text{GNF} \cup \text{GNE})^2 \rightarrow \text{GNF} \cup \text{GNE} \quad (\text{Composition of Normals and Neutrals}) \]

\[ ? \sqcap u = u \]
\[ u \sqcap ? = u \]
\[ x \sqcap x = x \]
\[ xp_1 u_1 \sqcap xp_2 u_2 = (xp_1 \sqcap xp_2)(u_1 \sqcap u_2) \]
\[ \langle \xi_1 \rangle N_1 \sqcap \langle \xi_2 \rangle N_2 = \langle \xi_1 \sqcap \xi_2 \rangle (N_1 \sqcap N_2) \]
\[ Type_{\ell} \sqcap Type_{\ell} = Type_{\ell} \]
\[ (x : U_1) \sqsubseteq U_2 \sqcap (x : U_1') \sqsubseteq U_2' = (x : U_1 \sqcap U_1') \sqsubseteq U_2 \sqcap U_2' \]
\[ \lambda x. u_1 \sqcap \lambda x. u_2 = \lambda x. u_1 \sqcap u_2 \]
\[ u_1 \sqcap u_2 = \text{undefined otherwise} \]

**Figure 3.20:** gdTL: Composing Normal Forms
For neutrals with neutral types (MCASTNEUTToUNKTAGNEUT) we must re-type them to account for possible differences between the two contexts. (Neutrals with non-neutral types are handled by MCASTNEUTFROMUNKToUNK and MCASTNEUTEvTYPE). Even when casting from ? to ? we must recursively cast subterms because of the possible differences in contexts (MCASTLAMFROMUNKToUNK, MCASTTYPEFROMUNKToUNK, MCASTPiFROMUNKToUNK, MCASTNEUTFROMUNKToUNK).

Likewise, each type former has rules to cast from ? (Fig. 3.19). Functions, function types, and universes are again cast by pushing casts to deeper subterms (MCASTLAMFROMUNK, MCASTPiFROMUNK, MCASTTYPEFROMUNK). For casting neutrals, if their type is ?, then their tag may also be ?, so we have two rules handling ?-tagged neutrals. The rule MCASTNEUTTAGUNKFROMUNKToTYPE casts a neutral with tag ? to $\text{Type}_i$, since this case is not handled by MCASTNEUTEvTYPE. In this case, the evidence $\langle \text{Type}_i \rangle$ captures (lack of) information from the type ? and the information from the new type $\text{Type}_i$, and succeeds as long as the neutral can be re-typed to a type consistent with $\text{Type}_i$ in the new context. The rule MCASTNEUTFROMUNKToPi handles the special case of casting a neutral from ? to a function type: since it is $\eta$-long at type ? but not at a function type, we re-type the neutral in the new context, then $\eta$-expand it applied to an $\eta$-expanded argument. Casting ?-typed neutrals with non-? tags is handled by MCASTNEUTToNEUT and MCASTNEUTEvTYPE.

### 3.5.4 Properties of Approximate Normalization

#### 3.5.4.1 Preservation of Typing

To prove type safety for $\text{gdtl}$, a key property of normalization is that it preserves typing. This property relies on the fact that hereditary substitution preserves typing, which can be shown using a technique similar to that of Pfenning [121].

**Theorem 3.5.1** (Normalization preserves typing). If $\Gamma \vdash u \leftarrow t \leftarrow U$, then $\Gamma \vdash u \leftarrow U$.

See proof in Appendix A.
3.5.4.2 Normalization Is Decidable

Since we have defined substitution and normalization using inference rules, they are technically relations rather than functions. The rules are syntax directed in terms of their inputs, so it is easy to show that there exists at most one result for every set of inputs. Since substitution and normalization are syntax directed and only have self-reference on strictly smaller sub-terms, we can decide whether a given term has a normal form when checked against some type.

**Theorem 3.5.2** (Normalization is Decidable). If $\Gamma \vdash t \ll U$, then $\Gamma \vdash u \leftrightarrow t \ll U$ for at most one $u$. Determining whether such a $u$ exists is decidable, as is computing it.

3.6 **GDTL: Run-time Semantics**

With the type system for GDTL realized, we turn to its dynamic semantics. Following the approaches of Garcia et al. [67] and Toro et al. [147], we let the syntactic type-safety proof for the static SDSL drive its design. In place of a cast calculus, gradual terms carry evidence matching their type, and computation steps evolve that evidence incrementally. When evidence no longer supports the well typedness of a term, execution fails with a run-time type error.

3.6.1 The Run-time Language

Figure 3.21 gives the syntax for the GDTL run-time language, which we call the evidence language. It mirrors the syntax for gradual terms, with two main changes. In place of type ascriptions is a special form for terms augmented with evidence, following Toro et al. [147]. As with compile-time evidence, run-time evidence is either a normal or neutral form. Notably, run-time terms are ascribed surface-normals as evidence, and evidence is normalized using approximate normalization, so all evidence computations terminate. We also have $U$, an explicit term for run-time type errors.

There is a straightforward elaboration from the surface language to the evidence language which augments the bidirectional typing rules to output the translated term. Type ascriptions are dropped in the GSyntahn rule, and initial evidence.
Evidence of consistency is added in \textsc{GCheckSynth}. Section 3.6.2 describes how to derive this initial evidence. In the \textsc{GSynthDyn} rule, we annotate \texttt{?} with evidence \texttt{?}, so \texttt{?} is always accompanied by some evidence of its type. Similarly, functions of type \texttt{?} are ascribed \texttt{(? \rightarrow ?)}.

In Fig. 3.21 we also define the class of syntactic values, which determines those terms that are done evaluating. We wish to let values be augmented with evidence, but not to have multiple evidence objects stacked on a value. To express this, we separate the class of values from the class of \emph{raw values}, which are never ascribed with evidence at the top level.
Values are similar to, but not the same, as normal forms. In particular, normal terms contain no redexes, even beneath a $\lambda$, whereas values may contain redexes within abstractions.

### 3.6.2 Typing and Evidence

To establish progress and preservation, we need typing rules for evidence terms, whose rules we give in Fig. 3.21. These rules are essentially the same as for gradual terms, with two major changes. First, we no longer use bidirectional typing, since our type system need not be syntax-directed to prove safety. The important properties are that each well typed surface term elaborates to a well typed evidence term, and that each reduction step preserves typedness for evidence terms. Second, whereas gradual terms could be given any type that is consistent with their actual type, we only allow this for dynamic terms directly ascribed with evidence, as seen in the rule $\text{EvTypeEv}$. Thus, all applications of consistency are made explicit in the syntax of evidence terms, and for a term $(\epsilon \, t)$, the evidence $\epsilon$ serves as a concrete witness between the actual type of $t$ and the type at which it is used. Unlike with normal forms, any evidence term can be ascribed with evidence. This difference reflects the different purposes for the two syntaxes. Normal forms are the target of normalization, where all possible reductions are performed, even under binders, so only neutrals need be ascribed with evidence, as the casting operation can traverse deep into terms. Evidence terms, model the run-time behaviour of gradual programs, where we do not reduce under binders, and reductions are performed sequentially, not all at once.

Because evidence ascriptions are normal forms, not evidence terms, we may need to substitute an evidence term into evidence, for example, when applying a function. To facilitate this, we extend the term normalization relation to operate on evidence terms by erasing evidence ascriptions. Using normal forms for evidence ensures that evidence computations terminate. We discuss the trade-offs of having terminating evidence more extensively in Chapters 6 and 7. We can then define hereditary substitutions of evidence terms into types, which is crucial when updating evidence after function applications.
Evidence has two key operations. First, we need initial evidence to elaborate gradual terms to evidence terms. If a term synthesizes some $U$ and is checked against $U'$, then during elaboration we can ascribe to it the evidence $\langle U \cap U' \rangle$. Secondly, we need a way to combine two pieces of evidence at run time, an operation referred to as consistent transitivity in AGT: if $\langle U \rangle \vdash U_1 \equiv U_2$, and $\langle U' \rangle \vdash U_2 \equiv U_3$, then $\langle U \cap U' \rangle \vdash U_1 \equiv U_3$, provided that the meet is defined. So we can also use the precision meet to dynamically combine different pieces of evidence.

Second, evidence is combined using the composition (meet) operation from Fig. 3.20, which is based on definitional (intensional) equality. If we have $A : (\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{Type}$, then $A(\lambda x.x + x - x)$ and $A(\lambda x.x)$ are not consistent at run time, despite being extensionally equivalent. There is a delicate balance between the decidability of consistency checking and dynamically respecting terms that are statically equivalent, which we examine in depth in Chapter 5.

3.6.3 Developing a Safe Semantics

As a driving force behind the design of our semantics, we imagine a hypothetical proof of progress and preservation. Progress tells us which expressions we need reduction rules for, and preservation tells us how to step while remaining well typed. We explain the rules here, but the full listing is given in Fig. 3.22.

3.6.3.1 Double Evidence

Since values do not contain terms of the form $\langle \varepsilon_2 \rangle (\langle \varepsilon_1 \rangle \text{rv})$, progress dictates that we need a reduction rule for such a case. If $\cdot \vdash \text{rv} : U$, then $\varepsilon_1 \vdash U \equiv U'$ and $\varepsilon_2 \vdash U' \equiv U''$, then $\varepsilon_1 \cap \varepsilon_2 \vdash U \equiv U''$, so we step to $\langle \varepsilon_1 \cap \varepsilon_2 \rangle \text{rv}$. If the meet is not defined, then a run-time error occurs.

3.6.3.2 Functions with Evidence

Two complications arise when reducing applications with evidence. The first is that in $\lambda x. t$, the variable $x$ may be free in evidence ascriptions within $t$. When performing a substitution, we need the type and normal form of the term replacing the variable. We use the notation $[x \Rightarrow t_1]^{wU}t_2$ to denote the syntactic replacement of $x$ by $t_1$ in $t_2$, where free occurrences of $x$ in evidence within $t_2$ are replaced
by \( u \) (the normal form of \( t_2 \)) using hereditary substitution at type \( U \). We use this operation to reduce applications.

A second issue is that, while the simple rules dictate how to evaluate a \( \lambda \)-term applied to a value, they do not determine how to proceed for applications of the form \((\langle U \rangle \lambda x. t) \ (\langle U' \rangle \text{rv})\). In such a case, we know that \( \cdot \vdash (\langle U \rangle \lambda x. t) : U_1 \) and that \( \langle U \rangle \vdash U_1 \models U_2 \) for some \( U_2 \). Computing \( \text{dom } U \cap U' \) either fails or yields evidence that the type of \( \text{rv} \) is consistent with the domain of \( U_1 \), so we ascribe this evidence during substitution to preserve well typedness. The evidence-typing rules say that the type of an application is found by normalizing the argument value and substituting into the codomain of the function type. To produce a result at this type, we can normalize \( \text{rv} \) and substitute it into \( \text{cod } \langle U \rangle \), thereby producing
evidence that the actual result is consistent with the return type. In the case where \( rv \) is not ascribed with evidence, we can behave as if it were ascribed \( ? \) and proceed using the above process.

### 3.6.3.3 Applying The Unknown Term

The syntax for values only admits application under binders, so we must somehow reduce terms of the form \((\varepsilon)?)v\). If the function is unknown, so is its output. Since the unknown term is always accompanied by evidence at run time, we calculate the result type by substituting the argument into the codomain of the evidence associated with \( ? \).

### 3.6.3.4 Remaining Cases

All other well typed terms are either values, or contain a redex as a subterm, either of the simple variety or of the varieties described above. Using contextual rules to account for these remaining cases, the AGT approach yields a semantics where type conversions satisfy progress and preservation by construction.

### 3.6.4 Example: Computing the head of the empty vector

We return to the example from Section 3.4.6, this time explaining its run-time behaviour. Because of consistency, the term \( \text{Nil} :: (\text{Vec Nat} ?) \) is given the evidence \( \langle \text{Vec Nat} ? \rangle \), obtained by computing \( \text{Vec Nat} 0 \cap \text{Vec Nat} ? \). Applying consistency to use this as an argument adds the evidence \( \langle \text{Vec Nat} 1 \rangle \), since we check \( \text{Nil} :: (\text{Vec Nat} ?) \) against \( \text{dom} (\text{Vec Nat} 1 \rightarrow \text{Nat}) \). The rule StepContext dictates that we must evaluate the argument to a function before evaluating the application itself. Our argument is \( \langle \text{Vec Nat} 1 \rangle \langle \text{Vec Nat} 0 \rangle \text{Nil} \), and since the meet of the evidence types is undefined, we step to \( \mathcal{U} \) with StepAscrFail.

### 3.7 Properties of GDTL

GDTL satisfies all the criteria for gradual languages set forth by Siek et al. [136].
3.7.1 Safety

First, GDTL is type safe by construction: it inherits safety for static operations from SDSL, and the evidence operations are specifically crafted to maintain progress and preservation. We can then obtain the standard safety result for gradual languages, namely that well typed terms do not get stuck.

**Theorem 3.7.1 (Type safety).** If \( \vdash t : U \), then either \( t \rightarrow^* v \) for some \( v \), \( t \rightarrow^* U \), or \( t \) diverges.

See proof in Appendix A.

So gradually well typed programs in GDTL may fail with run-time type errors, but they never get stuck. Among the three main approaches to deal with gradual types in the literature, GDTL follows the original approach of Siek and Taha [132] and Siek et al. [136], which enforces types eagerly at boundaries, including at higher-order types. This choice is in contrast with first-order enforcement (i.e. transient semantics [152]), or simple type erasure (also known as optional typing).

In particular, while the transient semantics supports open world soundness [152] when implemented on top of a (safe) dynamic language, it is unclear if and how this approach, which is restricted to checking type constructors, can scale to full-spectrum dependent types. GDTL is a sound gradually typed language that requires elaboration of the complete program to insert the pieces of evidence that support run-time checking.

3.7.2 Conservative Extension of SDSL

It is easy to show that GDTL is a conservative extension of SDSL, so any fully-precise GDTL programs enjoy the soundness and logical consistency properties that SDSL guarantees. Any statically typed term is well typed in GDTL by construction, thanks to AGT: on fully precise gradual types, \( \alpha \circ \gamma \) is the identity. Moreover, the only additions are those pertaining to ?, meaning that if we restrict ourselves to the static subset of terms (and types) without ?, then we obtain all the properties of the static system. We formalize the statement as follows:

\[ \text{Greenman and Felleisen [70] present a detailed comparative semantic account of these three approaches.} \]
Theorem 3.7.2. If $\Gamma_G, t_G, U_G$ are the embeddings of some $\Gamma_S, t_S, U_S$ into GDTL, and $\Gamma_G \vdash t_G \equiv U_G$, then $\Gamma_S \vdash t_S \equiv U_S$. Moreover, if $t_S \rightarrow^* v_S$ and $t_G$ elaborates to $t_E$, then there exists some $v_E$ where $t_E \rightarrow^* v_E$, where removing evidence from $v_E$ yields $v_S$.

3.7.3 Embedding of Untyped Lambda Calculus

A significant property of GDTL is that it can fully embed the untyped lambda calculus, including non-terminating terms. Given an untyped embedding function $[t]$ that (in essence) annotates all terms with $?$, we can show that any untyped term can be embedded in our system. Since no type information is present, all evidence objects are formed using $? \rightarrow$, and the meet operator never fails. So untyped programs behave normally in GDTL.

Theorem 3.7.3. For any untyped $\lambda$-term $t$ and closing environment $\Gamma$ that maps all variables to type $?$, then $\Gamma \vdash [t] \Rightarrow ?$. Moreover, if $t$ is closed, then $t \rightarrow^* v$ implies that $[t]$ elaborates to $t$ where $t \rightarrow^* v$ and stripping evidence from $v$ yields $v$.

See proof in Appendix A.

3.7.4 Gradual Guarantees

To state the gradual guarantees, we need a precision relation on contexts, which we give in Fig. 3.23, along with a syntactic formulation of precision that is equivalent to, but less opaque than $u_1 \sqsubseteq u_2 := \gamma(u_1) \subseteq \gamma(u_2)$. Precision on surface and evidence terms is defined in the exact same way: everything is at least as precise as the unknown term, and composite terms are precision-related if their parts are.

GDTL supports the gradual guarantees, with a smooth transition between dependent and untyped programming.

Theorem 3.7.4 (Gradual Guarantee).

(Static Guarantee) Suppose $\Gamma \vdash t \triangleleft U$ and $U \subseteq U'$. If $\Gamma \subseteq \Gamma'$ and $t \sqsubseteq t'$, then $\Gamma' \vdash t' \triangleleft U'$.

(Dynamic Guarantee) Suppose that $\cdot \vdash t_1 : U$, $\cdot \vdash t_1 : U'$, $t_1 \sqsubseteq t_2$, and $U \subseteq U'$. If $t_1 \rightarrow^* t_2$, then $t_1 \rightarrow^* t_2$ where $t_2 \sqsubseteq t_2$. 

108
See proof in Appendix A.

\(AGT\) ensures that the gradual guarantee holds by construction. Specifically, because approximate normalization and consistent transitivity are monotone with respect to precision, we can establish a weak bisimulation between the steps of the more and less precise versions [67].

A novel insight that arises from our work is that we need a restricted form of the dynamic gradual guarantee for normalization to prove the static gradual guarantee. To differentiate it from the standard one, we call it the \textit{normalization gradual guarantee}. Because an \(\eta\)-long term might be longer at a more precise type, we phrase the guarantee modulo \(\eta\)-equivalence: we say that \(U_1 \equiv^\eta U_2\) if \(U_1 =^\eta U_1'\), \(U_2 =^\eta U_2'\) and \(U_1' \equiv^\eta U_2'\).

With these defined, we can state the normalization gradual guarantee. We emphasize that while this theorem was claimed in [58], devising a version of GDTL that actually satisfies it is novel to this thesis.

**Lemma 3.7.5** (Normalization Gradual Guarantee). \textit{Suppose} \(\Gamma_1 \vdash u_1 \sim t_1 \Rightarrow U_1\). \textit{If} \(\Gamma_1 \equiv^\eta \Gamma_2, t_1 \equiv t_2,\) \textit{and} \(U_1 \equiv^\eta U_2,\) \textit{then} \(\Gamma_2 \vdash u_2 \sim t_2 \Rightarrow U_2\) \textit{where} \(u_1 \equiv^\eta u_2\).
3.8 Related Work

3.8.1 Static Dependent Types

The static dependently typed language SDTL, from which GDTL is derived, incorporates many features and techniques from the literature. The core of the language is very similar to that of $\text{cc}_\omega$ [39], albeit without an impredicative Prop sort, and the typed normalization judgment more closely resembles Martin-Löf Type Theory ($\text{MLTT}$). The core language of Idris [19], TT, also features cumulative universes with a single syntactic category for terms and types. Our use of canonical forms draws heavily from work on the Logical Framework (LF) [72, 73]. The bidirectional type system we adopt is inspired by the tutorial of Löh et al. [89]. Our formulation of hereditary substitution [121, 154] in SDTL is largely drawn from that of Harper and Licata [72], particularly the type-outputting judgment for substitution on neutral forms, and the treatment of the variable type as an extrinsic argument.

3.8.2 Mixing Dependent Types and Non-termination

Dependently typed languages that admit non-termination either give up on logical consistency altogether ($\text{Omega}$ [129], Haskell), or isolate a sublanguage of pure terminating expressions. This separation can be either enforced through the type system and/or a termination checker (Aura [82], $\text{F\#}$ [142], Idris [19]), or through a strict syntactic separation (Dependent ML [157], ATS [30]). The design space is large, reflecting a variety of sensible trade-offs between expressiveness, guarantees, and flexibility.

The Zombie language [23, 138] implements a flexible combination of programming and proving. The language is defined in two separate, but interacting, fragments: the programmatic fragment ensures type safety but not termination, and the logical fragment (a syntactic subset of the programmatic one) guarantees logical consistency. Programmers must declare in which fragment a given definition lives, but mobile types and cross-fragment case expressions allow interactions be-
tween the fragments. Zombie embodies a different trade-off from GDTL: while the logical fragment is consistent as a logic, type checking may diverge due to normalization of terms from the programmatic fragment. In contrast, GDTL eschews logical consistency as soon as imprecision is introduced (with ?), but approximate normalization ensures that type checking terminates.

3.8.3 Mixing Dependent Types and Simple Types

Several approaches [114, 144] for mixing dynamic and dependent typing have been proposed around the idea of subset types. This idea is extended by dependent interoperability [42, 43, 113], which allows the automatic lifting of subset-type based conversions to higher-order programs. These subset-type based approaches rely on the connection between pre-existing types, such as Vec and List, imposing the overhead of data structure conversions. The fact that List is a less precise structure than Vec is therefore defined a posteriori. In contrast, in GDTL, one can define List A as an alias for Vec A ?, thereby avoiding the need for deep structural conversions.

The work of Lehmann and Tanter [84] on gradual refinement types includes some form of dependency in types. Refinement types separate types from terms more cleanly, and hence do not need approximate normalization. However, they are less expressive than the dependent type system considered here. Furthermore, GDTL is the first gradual language to let ? be used in both term and type position, and to fully embed the untyped lambda calculus.

3.8.4 Programming with Holes

Finally, we observe that using ? in place of proof terms in GDTL is related to the concept of holes in dependently typed languages. Idris [19] and Agda [109] both allow type checking of programs with typed holes. The main difference between ? and holes in these languages is that applying a hole to a value results in a stuck term, while in GDTL, applying ? to a value produces another ?. Thus, when programming with holes, either undesired type errors occur, or type checks are ignored due to unsolved constraints.

Recently, Omar et al. [112] describe Hazelnut, a language and programming
system with typed holes that fully supports evaluation in presence of holes, including reduction around holes. The approach is based on Contextual Modal Type Theory [104]. It would be interesting to study whether the dependently typed version of CMTT [122] could be combined with the evaluation approach of Hazelnut, and the IDE support, to provide a rich programming experience with gradual dependent types.

3.9 Conclusion

GDTL represents a glimpse of the challenging and potentially large design space induced by combining dependent types and gradual typing. Specifically, this chapter proposes approximate normalization as a novel technique for designing gradual dependently typed languages, in a way that ensures decidable type checking and naturally satisfies the gradual guarantees.

Currently, GDTL lacks a number of features required of a practical dependently typed programming language. It might also be interesting to consider pattern matching as the primitive notion for eliminating inductives, as in Agda, instead of elimination principles as in Coq; the equalities implied by dependent matches could be turned into run-time checks for gradually typed values.

Future work includes supporting implicit arguments and higher-order unification, blame tracking [153], and efficient run-time semantics with erasure of computationally-irrelevant arguments [20]. Approximate normalization might be made more precise by exploiting termination contracts [107].
Chapter 4

Interlude: The Gradual Calculus of Inductive Constructions

In this chapter, we describe the Gradual Calculus of Inductive Constructions (GCIC). We emphasize that GCIC is not a contribution of this thesis: it was created by Lennon-Bertrand, Maillard, Tabareau, and Tanter [87]. In Section 4.1, we discuss the differences between the GCIC and GDTL approaches. The remaining text of this chapter is based on a section from the paper version of Chapter 5 [60], which presented GCIC as important background knowledge. We include it here, rather than in Chapter 2 or Chapter 5, for three reasons:

- Separating GCIC into its own chapter ensures that the distinction between previous work and novel contributions in Chapter 5 is clear;

- Chronologically, the paper presenting GCIC was published after the paper on which Chapter 3 is based [58], but before the paper on which Chapter 5 is based. So to understand the evolution of ideas surrounding gradual dependent types, familiarity with GDTL helps clarify the context under which GCIC was devised.

- Chapters 5 and 6 present extensions of GCIC, so it is not merely related to
our work, but is a critical piece of understanding our work.

To present gcic, we first describe the Bidirectional Calculus of Inductive Constructions (bcic), a modification of the Calculus of Inductive Constructions (cic) whose bidirectional types are convenient for gradual typing [86]. We then describe the gradual surface language gcic, along with Cast calculus for the Gradual Calculus of Inductive Constructions (castcic), and a translation from gcic to castcic [87]. Specifically, we use Gcic^G, the variant that satisfies the gradual guarantees and embeds cic, but sacrifices strong normalization. Chapter 6 is dedicated to regaining decidable type checking for an extension of gcic.

4.1 Overview: gdtl vs. gcic

In this section, we give a birds-eye view of the differences between the Gradually Dependent Typing Language (gdtl) and gcic, with the aim of providing context that helps the reader understand the details provided from Section 4.2 onward.

4.1.1 Similarities

gdtl and gcic are built on a similar foundation. They are both fully gradual, full spectrum dependently typed languages. Types and terms inhabit a single syntactic category, and ? can be used to replace a term of any type. Both languages implement sound gradual typing: deep run-time type information is kept to perform dynamic checks, as opposed to the shallow run-time type information of transient typing or the non-existent run-time type information of optional typing [70]. Neither language directly supports indexed families or propositional equality, though we extend gcic with these in Chapter 5.

4.1.2 Differences

4.1.2.1 Static Language

gdtl is variant of the Calculus of Constructions with a predicative universe hierarchy (ccw), whereas gcic is based on cic. That is, in gdtl the only static values
are functions, function types, and universes. GCIC has inductive types (but not indexed families), so users can define their own types as recursive sums of products. As such, GCIC is much closer to a model of a usable programming language with conditional branching and pattern matching.

GCIC also uses a very different flavour of bidirectional typing than the one used in GDTL. In GDTL, normal forms are checking, and neutral forms synthesize. Normal forms are typed using a relation similar to, but separate from, term typing. In GCIC, there is no separate judgment for typing of normal forms, which are a strict subset of terms. All syntactic forms are synthesizing, and the only checking rule is for conversion. Because of this, functions in GCIC must be ascribed with their argument types so that their typing rule can synthesize but still be syntax-directed.

The style of conversion check differs between GDTL and GCIC. In GDTL, conversion checking is done using type-directed judgmental equality, much like in Martin-Löf Type Theory (MLTT). However, unlike CIC or MLTT, judgmental equality compares terms by first normalizing them with hereditary substitution. In GCIC, convertibility of terms is defined using an untyped rewrite relation that can be applied to any subterm. This small-step style lends itself well to simulation arguments. Terms are judgmentally equal if there is some common term to which they both reduce.

4.1.2.2 The Unknown Term

In GDTL, the unknown term is never ascribed with its type: any program component can be replaced with ?. In contrast, GCIC requires that ? have a level ascription: if ? : T and T : Type, then ? is ascribed level ℓ. These ascriptions avoid the gradual versions of Girard’s paradox which can be expressed in GDTL.

4.1.2.3 Dynamic Type Checks

GDTL is based on Abstracting Gradual Typing (AGT), whereas GCIC uses the cast calculus approach. In GDTL’s run-time language, all terms (including values) may be ascribed with evidence of the consistency between their actual type and the type at which they are used. Run-time checks are performed by composing
stacked evidence, and failures happen when the composition of such evidence is not defined. In $\text{GCIC}$, type imprecision is expressed computationally: the run–time language has an explicit form for casts. Values can only contain casts if they are neutral, or if the casts are on $?^1$. Type information is not explicitly composed, but casts are pushed inward in terms, and they fail if the source and destination have inconsistent top-level constructors.

The two languages also differ in their handling of errors and function types. In $\text{GDTL}$, function type information is eagerly composed, so using a function of type $\text{Int} \rightarrow \text{Int}$ in a context expecting $\text{Bool} \rightarrow \text{Bool}$ fails. In $\text{GCIC}$, casts are lazily applied to function types: casting from $\text{Int} \rightarrow \text{Int}$ to $\text{Bool} \rightarrow \text{Bool}$ succeeds but produces a function that always raises an error when called. This is fitting for $\text{GCIC}$, however, as the error $\Upsilon$ at type $(x : A) \rightarrow B$ is $\eta$-expanded to $\lambda(x : A).\Upsilon$.

4.1.2.4 Normalization and the Gradual Guarantees

Decidable type checking is the driving force behind $\text{GDTL}$: its syntax, typing, and semantics are all designed around the goal of ensuring all type computations terminate. As such, different semantics are used for normalizing during type checking and executing at run time. The core calculus is limited to ensure that hereditary substitution can be defined using well founded recursion.

$\text{GCIC}$ is instead designed around the aim of providing a fully gradual type theory. Compile-time and run-time computations use the exact same reduction rules, and the mechanism for prevent non-termination is to reject such terms, rather than to approximate them. The original presentation of $\text{GCIC}$ explores the inherent tradeoffs with strong normalization. The authors present three variants of $\text{GCIC}$. $\text{GCIC}^G$ behaves much like the run-time language of $\text{GDTL}$, satisfying the gradual guarantees and allowing non-termination, at the expense of strong normalization and decidable type checking. $\text{GCIC}^N$ contains an extra universe restriction that allows terms to raise errors instead of diverging, enabling strong normalization at

---

$^1$Casts on $?$ are really an abuse of notation. We would prefer to separate the computation of converting from $?$ of one type to another from the act of tagging $?$ with the type of values it could plausibly represent, and have casts on $?$ reduce to tagged versions of $?$ by composing the cast type with the information in the tag. But we follow the original presentation of $\text{GCIC}$ in this case, which does help reduce the notational overhead.
the cost of the gradual guarantees. Finally, \( \text{gcic}^\dagger \) obtains both strong normalization and the gradual guarantees by statically restricting the universes over which function types can quantify. However, this restriction means there are static CIC programs that cannot be typed in \( \text{gcic}^\dagger \).

Lennon-Bertrand et al. [87] show that no language can conservatively extend the Simply Typed Lambda Calculus (STLC), have all terms strongly normalizing, and have the embedding-projection pairs (EP-pairs) property (Section 2.1.2.3). The three \( \text{gcic} \) variants each sacrifice one property to obtain the other two. However, \( \text{gdtl} \) is able to conservatively extend STLC while satisfying decidable type checking and the gradual guarantees (but not EP-pairs), something no variant of \( \text{gcic} \) is able to do. Chapter 6 shows how to extend approximate normalization to a language that includes inductive types.

### 4.2 The Bidirectional CIC

#### 4.2.1 Syntax

Figure 4.1 gives the bidirectional calculus of inductive constructions (bcic), which was created by Lennon-Bertrand [86], though we modify their notation to fit with the rest of this thesis. \( \text{bcic} \) terms are denoted by metavariables \( t \) and \( T \), loosely following the convention that \( T \) be reserved for types. Variables are denoted by \( x, y, z \). The notation \( \overrightarrow{V} \) denotes a sequence of objects matching metavariable \( V \). \( \text{bcic} \) has variables, a predicative hierarchy of universes, function types, functions, and applications. Technically, \( \text{bcic} \) extends the predicative, non-cumulative fragment of \( \text{cic} \): each function type is in a higher universe than its domain and codomain, and there is no subtyping between universe levels. We assume a pre-
existing set of inductive type constructors, denoted by the metavariable $C$, each of which has a fixed set of data constructors $D^C$. Type and data constructors are annotated with the universe level of their type, though we omit these annotations when they are not relevant.

To eliminate members of inductive types, a combined form $\text{ind}_C(t_1, z.T, x.y.t_2)$ replaces CIC’s fix and match. This form branches on the scrutinee $t_1$ and has a parameterized result type $T$, called the motive [96], that binds a variable $z$ of the scrutinee’s type, giving the result type of elimination for a particular input $z$. The branches $\overrightarrow{t_2}$ correspond to the constructors $\overrightarrow{D^C}$ of $C$. In each branch, the variables $\overrightarrow{y}$ are bound to the arguments to $D^C$, and $x$ is bound to the whole $\text{ind}_C$ expression, to facilitate recursion. The $\text{ind}_C$ form expresses an induction principle: if each branch produces a result of type $T$ where $z$ is bound to $D^C$ applied to $\overrightarrow{y}$, the elimination has type $T$ where $z$ is bound to the scrutinee $t_1$. In essence, $\text{ind}_C$ says that if we can build a $T$ for each constructor $D^C$ of $C$ with the given parameters, then we can build one for any value of $C$ with those parameters. Normally, a separate check ensures that recursive calls are only made on structurally smaller arguments, but we omit this check, since it is orthogonal to gradual typing and GCIC$^G$ would not be strongly normalizing even with it.

BCIC also uses head tags, denoted by $h$, which act as symbols to specify a type constructor without specifying its arguments. We use these in typing, e.g. for expressing that an applied function must synthesize a function type, even though we do not know what the domain and codomain should be. Head tags (or tags for short) are also useful in GCIC for defining the least precise type with a given head.

### 4.2.2 Typing and Semantics

The semantics of BCIC (Fig. 4.2) is standard, and is given with primitive notions of reduction $\rightarrow$, contextual stepping $\longrightarrow$ where any sub-term reduces (even under binders), and multi-step contextual reduction $\longrightarrow^*$, which allows zero or more steps with $\longrightarrow$.

The typing (Fig. 4.3) for BCIC resembles the typical presentation of CIC, but typing is divided into synthesis, which produces a type, and checking, which consumes a type. Because function types bind their parameter in the codomain,
applications synthesize a type depending on the value of the argument, since it is substituted for \( x \) in the codomain type. A term checks against any type that reduces to the same type as its synthesized type, since an application may have produced a type that must be reduced before comparing. Constrained-synthesis, \( \Gamma \vdash t \Rightarrow h T \), generalizes the pattern of synthesizing a type for a term after reducing it to a point that it has the desired head \( h \). Separating synthesis from constrained synthesis will be useful once gradual types are introduced. Function application has the standard \( \beta \)-reduction rule.

For inductive types, the typing rule establishes that, if \( T_P \) is a type parameterized over a value \( x \) from the inductive type \( C \), and we can (recursively) build a \( T_P \) for each constructor of \( C \), then we can build a \( T_P \) for any member or \( C \). Hence, \( \text{ind}_C() \) form gives an induction principle for \( C \), hence the notation \( \text{ind}_C(\ldots) \). Inductive types may be parameterized, but each constructor has the same return type. The reduction rule says that an \( \text{ind}_C(\ldots) \) form given a value \( D^C(\ldots) \) reduces to the branch corresponding to \( D^C \).
Figure 4.3: Bidirectional CIC: Syntax, Typing and Semantics
4.3 gcic: The Surface Language

Figure 4.4 extends BcIC into gcic, the Gradual CIC, by adding the imprecise term \( ?_{@i} \), which can be used at any type in universe level \( i \), along with type ascriptions, which were not in BcIC because all forms synthesized types. We use \( ?_T \) as sugar for \( ?_{@i} \) when \( T : Type_i \).

Dependent types complicate the typing of gcic. Because the dynamic semantics of gcic are defined using a cast calculus, and typing refers to reduction of terms, Lennon-Bertrand et al. [87] define gcic typing with cast calculus types. Nevertheless, we can establish lemmas (Fig. 4.4), phrased like rules, which provide intuition for how gcic terms are typed against gcic types, helping gcic be understood without diving into the details of the cast calculus.

The unknown term \( ?_{@i} \) synthesizes \( ?_{@i+1} \), i.e. its type is unknown, one level up in the universe hierarchy. A term checks against any type consistent with its synthesized type, where the relation \( \equiv_{\rightarrow} \) is understood to mean convertibility up to well typed occurrences of \( ? \). An ascribed term synthesizes the given type if it checks against it, relaxing or tightening the types of gradual terms. In the case where a term synthesizes \( ?_{Type_i} \), constrained synthesis produces the germ\(^2\).

The type germ\(_i(h)\) is the least precise type with a given head in universe \( i \). For function types, the germ is \( ?_{@i} \rightarrow ?_{@i} \), and \( Type_i \) is its own germ. For inductives, the germ is \( C(\text{Param}(\_)) \) where the \( i \)'s are the parameters’ levels.

\(^2\)called the ground type in non-dependently typed literature

\[
\begin{align*}
\Gamma \vdash t \equiv ?_{@i} \mid t :: T & \quad \Gamma \vdash ?_{@i} \Rightarrow ?_{@i+1} & \quad \Gamma \vdash t \leftarrow T \Rightarrow T \\
\frac{\Gamma \vdash t \Rightarrow T' \quad T' \equiv \rightarrow T}{\Gamma \vdash t \leftarrow T} & \quad \frac{\Gamma \vdash t \Rightarrow T \quad T \rightarrow^* ?_{@i}}{\Gamma \vdash ?_{@i} \rightarrow ?_{@i+1}} & \quad \frac{\Gamma \vdash t \Rightarrow \Pi ?_{@i} \rightarrow ?_{@i}}{\Gamma \vdash t \Rightarrow \Pi ?_{@i} \rightarrow ?_{@i+1}} \\
\frac{\Gamma \vdash t \Rightarrow Type \quad T \rightarrow^* ?_{@i}}{\Gamma \vdash t \Rightarrow C(\text{Param}(\_))} & \quad \frac{\Gamma \vdash t \Rightarrow Type \quad T \rightarrow^* ?_{@i+1}}{\Gamma \vdash t \Rightarrow Type_i}
\end{align*}
\]

Figure 4.4: gcic: Syntax and Typing Lemmas
4.4 **CastCIC: The Cast Calculus**

4.4.1 Syntax, Typing and Reductions

Figure 4.5 presents the added syntax and typing for CastCIC, the cast calculus for gcic. CastCIC extends bcic with the unknown term ?, an error \( \mathcal{U} \), and a cast \( \langle T_2 \leftarrow T_1 \rangle t \) from type \( T_1 \) to \( T_2 \). Forms \( ?_T \) and \( \mathcal{U}_T \) are ascribed with their type \( T \), which affect the dynamic semantics of CastCIC. In the new rules, terms \( ?_T \) and \( \mathcal{U}_T \) synthesize their ascribed type \( T \), while casts synthesize the destination type, given that the term being cast checks against the source type, and that both types are well-formed (CastCast). Because casts are explicit, the rule CastCheck uses definitional equality, rather than consistency, when switching from synthesis to checking.

The CastCIC semantics includes all bcic reductions, plus cast rules (Fig. 4.7) and propagation rules (Fig. 4.6). The propagation rules handle ? and \( \mathcal{U} \). At type \( (x : T_1) \to T_2, ? \) and \( \mathcal{U} \) expand to \( \lambda (x : T_1). ?_T \) in RedPropFunUnk and \( \lambda (x : T_1). \mathcal{U}_T \) in RedPropFunErr. In the remaining RedProp rules, eliminating or casting ? or \( \mathcal{U} \) produces ? or \( \mathcal{U} \).

Cast rules either convert between types with the same head, cast to \( ?_{\text{Type}_1} \), or produce an error. A cast from \( \text{Type}_1 \) to itself reduces away (RedCastType). For inductives, casts from \( \overrightarrow{C(T_1)} \) to \( \overrightarrow{C(T_2)} \) reduce by casting fields to the new type given by the different parameter values (RedCastInd). Note that \( t_{\text{par}} \) and \( t'_{\text{par}} \) need not match, but typing guarantees that they are convertible. Casts between types with mismatched heads produce an error (RedCastHeadErr), as do casts to or from \( \mathcal{U}_{\text{Type}_1} \) (RedCastDomErr, RedCastCodomErr). A cast from the germ for a given head does not reduce: \( \langle ?_{\text{Type}_1} \leftarrow \text{germ}(T) \rangle \) acts as a tag, injecting into \( ?_{\text{Type}_1} \). Casts from non-germ types to \( ?_{\text{Type}_1} \) decompose into casts through the germ that are then tagged with their injection into \( ?_{\text{Type}_1} \) (RedCastFunGerm, RedCastIndGerm). In RedCastUpDown, a cast from \( ?_{\text{Type}_1} \) to \( T \) reduces when the value being cast originates from a type with a matching head, and was accordingly tagged with a cast from head(T) to \( ?_{\text{Type}_1} \).
Figure 4.5: CastCIC: Typing and New Syntax
### 4.4.2 Elaboration

Finally, elaboration (Figs. 4.8 and 4.9) defines the relationship between GCIC and CastCIC, which in turn defines which GCIC terms are well typed. Like CastCIC, elaboration has synthesis, checking, and constrained synthesis, but each additionally produces the elaboration of the subject term as output, so it can be used to compute the return types of dependent functions and eliminations. The rule ElabUnk synthesizes the unknown type for the unknown term at the given universe level. ElabApp works like a normal dependent function application, but uses the elaboration of the argument to replace the parameter in the return type. ElabCst checks a term against a type consistent with its synthesized type, inserting the cast between these types into the elaboration. Figure 4.8 also defines new constrained synthesis rules. Rule ElabUnkFun produces an elaboration where the applied value of type ?Type is cast to a function type, so it can be safely applied. Rules ElabUnkInd and ElabUnkUniv work similarly. We omit the elaboration rules corresponding to the remaining BCIC rules, which homomorphically elaborate the sub-terms of a given term, but refer the curious reader to the original presentation [87].

Consistency in GCIC is checked modulo conversion, with a relation defined in Fig. 4.10. Two terms are consistently convertible if they step to terms that are α-consistent. The α-consistency relation (Fig. 4.10) is very similar to consistency in GDL: every term is consistent with ?, and terms with the same head are con-
Cast Reductions:

\[
\begin{align*}
\text{RedCastUpDown} & \\
\langle T \leftarrow ?\text{Type}_i \rangle \langle ?\text{Type}_i \leftarrow \text{germ}_i(h) \rangle t \rightsquigarrow \langle T \leftarrow \text{germ}_i(h) \rangle t
\end{align*}
\]

\[
\begin{align*}
\text{RedCastInd} & \\
\langle C(l'_\text{par}) \leftarrow C(l_{\text{par}}) \rangle \overrightarrow{\mathcal{D}}(l_{\text{par}^n}, t_{\text{arg}}) \rightsquigarrow \overrightarrow{\mathcal{D}}(l'_\text{par}, t'_{\text{arg}})
\end{align*}
\]

where \( t'_{\text{arg}} := \langle \text{ Args}_i(C, i, D) \rangle(t'_\text{par}, t'_{\text{arg}}) \to \langle \text{ Args}_i(C, i, D) \rangle t_{\text{arg}} \).

\[
\begin{align*}
\text{RedCastType} & \\
\langle \text{Type}_i \leftarrow \text{Type}_j \rangle T \rightsquigarrow T
\end{align*}
\]

\[
\begin{align*}
\text{RedCastDomErr} & \\
\langle T \leftarrow U_{\text{Type}_i} \rangle T \rightsquigarrow U_T
\end{align*}
\]

\[
\begin{align*}
\text{RedCastCodomErr} & \\
\langle U_{\text{Type}_i} \leftarrow T \rangle T \rightsquigarrow U_{U_{\text{Type}_i}}
\end{align*}
\]

\[
\begin{align*}
\text{RedCastHeadErr} & \\
\text{head}(T) \neq \text{head}(T') \\
\langle T' \leftarrow T \rangle T \rightsquigarrow U_T
\end{align*}
\]

\[
\begin{align*}
\text{RedCastFunGerm} & \\
\langle x : T_1 \to T_2 \rangle \to T_0 \neq \text{germ}_j(\Pi) \text{ for } j \geq i
\end{align*}
\]

\[
\begin{align*}
\langle ?\text{Type}_i \leftarrow (x : T_1) \to T_2 \rangle t \rightsquigarrow \langle ?\text{Type}_i \leftarrow ?\text{Type}_i \to ?\text{Type}_i \rangle \langle ?\text{Type}_i \to ?\text{Type}_i \leftarrow (x : T_1) \to T_2 \rangle t
\end{align*}
\]

\[
\begin{align*}
\text{RedCastIndGerm} & \\
\overrightarrow{\mathcal{C}(l_{\text{par}})} \neq \text{germ}_j(C) \text{ for } j \geq i
\end{align*}
\]

\[
\begin{align*}
\langle ?\text{Type}_i \leftarrow \overrightarrow{\mathcal{C}(l_{\text{par}})} \rangle t \rightsquigarrow \langle ?\text{Type}_i \leftarrow \overrightarrow{\mathcal{C}(?\text{Params}_i(C))} \rangle \langle ?\text{Type}_i \leftarrow \overrightarrow{\mathcal{C}(?\text{Params}_i(C))} \rangle \to \overrightarrow{\mathcal{C}(l_{\text{par}})} \rangle t
\end{align*}
\]

\[
\begin{align*}
\text{RedCastSizeErr} & \\
\min \{ j \mid \exists h. \text{germ}_i(h) = T \} > i
\end{align*}
\]

\[
\begin{align*}
\langle ?\text{Type}_i \leftarrow T \rangle t \rightsquigarrow U_{\text{Type}_i}
\end{align*}
\]

Figure 4.7: CASTCIC: Cast Reduction Rules

consistent if their parts are. Additionally, terms with casts are consistent are checked for consistency by ignoring the casts. The name \( \alpha \)-consistency comes from the fact that it only considers consistency up to (implicit) \( \alpha \)-equivalence, not modulo conversion.

Elaboration defines GCIC typing for surface terms and types: we say \( t : T \) when \( \cdot \vdash T \to T \Rightarrow_{\text{Type}} \text{Type}_i \) and \( \cdot \vdash t \to t \Leftarrow T \).
\[
\begin{align*}
\Gamma \vdash t \rightarrow t \Rightarrow T & \quad \Gamma \vdash t \rightarrow t \Leftarrow T & \quad \Gamma \vdash t \rightarrow t \Rightarrow h T \quad \text{(Elaboration)} \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| ElabCst | \( \Gamma \vdash t \rightarrow t \Rightarrow T' \)  
\( T' \cong \_ \_ T \) |
| ElabUnk | \( \Gamma \vdash ?@i \rightarrow ?\text{Type}_i \Rightarrow ?\text{Type}_i \)  
\( \Gamma \vdash t \rightarrow (T \Leftarrow T')t \Leftarrow T \) |
| ElabUnkFun | \( \Gamma \vdash t \rightarrow ?\text{Type}_i \)  
\( \Gamma \vdash t \rightarrow (\text{Type}_i \rightarrow ?\text{Type}_i \Leftarrow ?\text{Type}_i) \Rightarrow \Pi ?\text{Type}_i \rightarrow ?\text{Type}_i \) |
| ElabApp | \( \Gamma \vdash t_0 \rightarrow t_0 \Rightarrow \Pi (x : T_1) \rightarrow T_2 \)  
\( \Gamma \vdash t_1 \rightarrow t_1 \Leftarrow T_1 \)  
\( \Gamma \vdash t_0 \rightarrow t_0 \rightarrow [t_1/x]T_2 \)  
\( \Gamma \vdash \text{Type}_i \rightarrow \text{Type}_i \Rightarrow \text{Type}_{i+1} \) |
| ElabType | \( \Gamma \vdash t \rightarrow ?_i \)  
\( \Gamma \vdash t \rightarrow (C(\text{Params}(C,i)) \Leftarrow ?_i) \Rightarrow C(\text{Params}(C,i)) \) |
| ElabUnkUniv | \( \Gamma \vdash t \rightarrow ?\text{Type}_{i+1} \)  
\( \Gamma \vdash t \rightarrow (\text{Type}_i \Leftarrow ?\text{Type}_{i+1}) \Rightarrow \text{Type}_i \)  
\( \Gamma \vdash \text{Type}_i \rightarrow \text{Type}_i \Rightarrow \text{Type}_{i+1} \)  
\( \Gamma \vdash x \rightarrow x \Rightarrow T \) |
| ElabVar | \( \Gamma \vdash x \rightarrow x \Rightarrow T \) |
| ElabFun | \( \Gamma \vdash T_i \rightarrow T_i \Rightarrow \text{Type}_{i} \)  
\( \Gamma, x : T_1 \vdash T_2 \Rightarrow \text{Type}_{j} \)  
\( \Gamma \vdash (x : T_1) \rightarrow T_2 \rightarrow (x : T_1) \rightarrow T_2 \Rightarrow \text{Type}_{\text{max}(i,j)} \) |
| ElabAbs | \( \Gamma \vdash T_i \rightarrow T_i \Rightarrow \text{Type}_{i} \)  
\( \Gamma, x : T_1 \vdash t \Rightarrow T_2 \)  
\( \Gamma \vdash \lambda (x : T_1).t \rightarrow \lambda (x : T_1).t \Rightarrow (x : T_1) \rightarrow T_2 \) |

**Figure 4.8:** Elaboration from gcic to CastCIC
\[ \frac{\Gamma \vdash t_k \rightarrow t_k \Leftarrow \text{Params}_k(C,i)[t]}{\Gamma \vdash C_{@i}(\overrightarrow{t}) \rightarrow C_{@i}(\overrightarrow{t}) \Rightarrow \text{Type}_i} \]

\[ \frac{\Gamma \vdash t_k \rightarrow t_k \Leftarrow \text{Params}_k(C,i)[t]}{\Gamma \vdash t'_{m} \rightarrow t'_{m} \Leftarrow \text{Args}_m(C,i,D)[t',t']} \]

\[ \frac{\Gamma \vdash D_{C_{@i}}(\overrightarrow{t},\overrightarrow{t'}) \rightarrow D_{C_{@i}}(\overrightarrow{t},\overrightarrow{t'}) \Rightarrow C_{@i}(\overrightarrow{t}) \Rightarrow }{\Gamma \vdash \text{ind}_C(t_{\text{ scrut}},z.TP,x_{\text{rec}},y.t_{\text{ rhs}}) \rightarrow \text{ind}_C(t_{\text{ scrut}},z.TP,x_{\text{rec}},y.t_{\text{ rhs}}) \Rightarrow [t_{\text{ scrut}}/z]TP} \]

**ElabConstrSynthType**

\[ \frac{\Gamma \vdash t \twoheadrightarrow T}{T \twoheadrightarrow \ast \text{ Type}_i} \]

\[ \frac{\Gamma \vdash t \twoheadrightarrow T}{\Gamma \vdash t \twoheadrightarrow T \Rightarrow Type \text{ Type}_i} \]

**ElabConstrSynthFun**

\[ \frac{\Gamma \vdash t \twoheadrightarrow T \Rightarrow Type \text{ Type}_i}{\Gamma \vdash t \twoheadrightarrow T \Rightarrow \Pi (x : T_1) \rightarrow T_2} \]

\[ \frac{\Gamma \vdash t \twoheadrightarrow T \Rightarrow Type \text{ Type}_i}{\Gamma \vdash t \twoheadrightarrow T \Rightarrow Type \text{ Type}_i} \]

**ElabConstrSynthInd**

\[ \frac{\Gamma \vdash t \twoheadrightarrow T}{T \twoheadrightarrow \ast C_{@i}(t_p)} \]

**Figure 4.9:** Elaboration from GCIC to CASTCIC (ctd.)
Consistent Convertibility:

\[
\begin{align*}
\text{ConvCst} & : T_1 \rightarrow^* T'_1 \quad T_2 \rightarrow^* T'_2 \quad T'_1 \equiv_{\alpha} T'_2 \\
& \quad \Rightarrow T_1 \equiv_{\alpha} T_2
\end{align*}
\]

\(\alpha\)-Consistency:

\[
\begin{align*}
\text{CstVAR} & : x \equiv_{\alpha} x \\
\text{CstUnkL} & : \ ?_1 \equiv_{\alpha} t \\
\text{CstUnkR} & : t \equiv_{\alpha} ?_2 \\
\text{CstCastL} & : t \equiv_{\alpha} t' \Rightarrow \langle T_2 \leftarrow T_1 \rangle t \equiv_{\alpha} t' \\
\text{CstCastR} & : t \equiv_{\alpha} t' \Rightarrow \langle T_2 \leftarrow T_1 \rangle t \equiv_{\alpha} t'
\end{align*}
\]

\[
\begin{align*}
\text{CstAbs} & : T_1 \equiv_{\alpha} T_2 \\
& \quad \Rightarrow \lambda(x : T_1). t_1 \equiv_{\alpha} \lambda(x : T_2). t_2 \\
\text{CstCast} & : T_1 \equiv_{\alpha} T_2 \\
& \quad \Rightarrow \langle T'_1 \leftarrow T_1 \rangle t_1 \equiv_{\alpha} \langle T'_2 \leftarrow T_2 \rangle t_2
\end{align*}
\]

\[
\begin{align*}
\text{CstInd} & : C_{\oplus(i)} t_1 \equiv_{\alpha} C_{\oplus(i)} t_2 \\
\text{CstCons} & : t_1 \equiv_{\alpha} t_2 \\
\text{CstAPP} & : t_1 \equiv_{\alpha} t_2 \\
& \quad \Rightarrow \langle t'_1 \leftarrow t_1 \rangle t'_2 \equiv_{\alpha} t'_1 \equiv_{\alpha} t'_2
\end{align*}
\]

\[
\begin{align*}
\text{CstProd} & : \text{Type}_\ell \equiv_{\alpha} \text{Type}_\ell \\
& \quad \Rightarrow (x : T_1) \rightarrow T'_1 \equiv_{\alpha} (x : T_2) \rightarrow T'_2
\end{align*}
\]

\[
\begin{align*}
\text{CstFix} & : t \equiv_{\alpha} t' \\
& \quad \Rightarrow \text{ind}_C(t) \equiv_{\alpha} \text{ind}_C(t')
\end{align*}
\]

**Figure 4.10:** CstCic Consistency and Consistent Convertibility

### 4.4.2.1 A Note on Precision

We defer the presentation of GCIC’s definition of precision to Chapter 5. Precision is not needed to define GCIC itself, only to state the gradual guarantees for GCIC. Moreover, we make a small but pervasive change to the definition of precision in Chapter 5, so we present it there instead of presenting two nearly-identical definitions separately.
Chapter 5

Propositional Equality and Dynamic Consistency Tracking

In this chapter, we add propositional equality to a gradual dependently typed language. In the previous chapters, we saw how gradual dependent types can potentially help with the incremental adoption of dependently typed code. The given theories, however, lack propositional equality, which is a central feature of type theory.

The static equality type cannot be added to a gradual language without significant changes. Lennon-Bertrand et al. [87] show that, when the reflexive proof \( \text{refl} \) is the only closed value of an equality type, any gradual extension of the Calculus of Inductive Constructions (CIC) with propositional equality violates static observational equivalences. Extensionally-equal functions should be indistinguishable at run time, but they can be dynamically distinguished using a combination of equality and type imprecision. Violating static equivalence is problematic, because optimizations, program transformations, or reasoning principles that were statically valid may not be gradually valid.

Here, we present a notion of gradual propositional equality where static equivalences are respected. We do so by devising an equality type of which \( \text{refl} \) is not the only closed inhabitant. Instead, each equality proof is accompanied by a term that is at least as precise as the equated terms, acting as a witness of their plausible equality. These witnesses track partial type information as a program runs,
raising errors when that information shows that two equated terms are undeniably inconsistent. Composition of type information is internalized as a construct of the language, and is deferred for function bodies whose evaluation is blocked by variables. We thus ensure that extensionally-equal functions compose without error, thereby preventing contexts from distinguishing them. We describe the challenges of designing consistency and precision relations for this system, along with solutions to these challenges. Finally, we prove important metatheory: type safety, conservative embedding of the Calculus of Inductive Constructions (CIC), weak canonicity, and the gradual guarantees, which ensure that reducing a program’s precision introduces no new static or dynamic errors.

5.1 Gradual Propositional Equality

The propositional equality \[ \text{prop \ equality} \] \cite{94} type, written \( t_1 \equiv_T t_2 \) expresses that \( t_1 \) and \( t_2 \) are equal inhabitants of type \( T \). Its only constructor is \( \text{refl} : t \equiv_T t \), the proof that every term is equal to itself. Equality is useful for practical dependently typed programming, since it lets a function express pre- and postconditions by taking or returning equality proofs. Likewise, a programmer can use an equality proof to rewrite the type of the expression they are trying to produce. Propositional equality even lays a path to support GADT-style inductive families, since constructors with different return types can be encoded with non-indexed inductive types and propositional equality \cite{95}.

Propositional equality is immensely useful for programming, but it has been largely omitted from gradual languages, where only limited means of representing and reasoning about equality have been available. In the Gradual Calculus of Inductive Constructions (GCIC) \cite{87}, decidable equality is supported (see Section 5.2.2), where a type is computed by pattern-matching on the equated terms. Gradual Refinement Types \cite{84} support first-order constraints in linear integer arithmetic. In contrast, propositional equality is more general than refinement types and more lightweight than decidable equality: it works for every type, provides its own construction and elimination principles, and can be used with quantifiers or higher order functions.

One challenge with gradual equality has been propagating and enforcing equal-
ity constraints at run time. The problem is that equated terms may contain functions or dependent function types, both of which bind variables. For example, 
\((\lambda x. x + 0) \equiv_N (\lambda x. x)\) and 
\(\{(x : N) \rightarrow Vec N (x + 0)) \equiv_{Type} ((x : N) \rightarrow Vec N x)\) are well-formed types. Extensional equality of functions, even up to partial information, is undecidable. Comparing functions syntactically, by directly comparing bound variables, is decidable. Such a notion works for compile-time consistency checks, but would be problematic during run-time checks, since it destroys static reasoning principles. Observationally equivalent terms in the static language could be distinguishable in the gradual language, because a dynamic type check can flag them as being syntactically different e.g., replacing \(\lambda x. x + x\) with \(\lambda x. 2 * x\) can cause new dynamic errors. A language that dynamically compares terms with bound variables is impractical: it cannot be easily compiled to efficient code, since every function now needs a syntactic representation that persists at run time. Even worse, Lennon-Bertrand et al. [87] show that when \(\texttt{refl}\) is the only static constructor for propositional equality, it is not possible to include said equality type in a gradual language while respecting static equivalences.

Our key insight is to represent an equality proof using a witness that captures equality constraints discovered at run time. We do so with a Gradual language with propositional Equality (\(\texttt{GEQ}\)), pronounced “geek”, which adds propositional equality to \(\texttt{gsc}{}\), allowing \(=, \texttt{refl}\) and \(\texttt{J}\) to be used like in the Calculus of Inductive Constructions (CIC), but with a dynamic semantics that is meaningful for gradual types. Taking inspiration from evidence in Abstracting Gradual Typing (\(\texttt{agt}\)) [67] and middle-types in threesomes [135], we represent witnesses with a term that is as precise as both equated terms. As a consequence, \(\texttt{refl}\) and \(\texttt{J}\) are not the only inhabitants of the equality type, avoiding the above impossibility result.

This chapter is structured as follows:

- We demonstrate how equality proofs between imprecise terms are useful for discovering bugs in programs and for guiding the development of static proofs (Section 5.2);
- We extend \(\texttt{gsc}\) with propositional equality (Section 5.3) by typing equality using consistency witnesses between terms (Section 5.3.2). We give operational semantics via a cast calculus, where the eliminator for equality uses casts go-
ing through the result type given by the witness (Section 5.3.3). To combine witnesses when casting between equality types, we add witness composition directly as a construct in GEq (Section 5.3.4). This operator delays the comparison of neutral terms until their variables are bound to values, so composing statically equivalent functions does not raise an error;

• We prove type safety, conservative extension of cic, weak canonicity, and the gradual guarantees for GEq (Section 5.4), so imprecision never causes stuck states or new (static or dynamic) errors, and GEq rejects ill typed cic programs. Like Siek and Chen [131], our proofs are parameterized over definitions of consistency and precision, revealing sufficient properties to prove the theorems;

• We define precision and consistency for the cast calculus (Section 5.5), showing that they fulfill the previously identified properties. We separate static consistency, whether a term of some type can be used in a given context, from dynamic consistency, whether two terms compose without error. These coincide for non-dependent gradual languages, but in GEq they must be separated to respect static equivalences while still rejecting ill typed static programs.

• Section 5.6 discusses extensions enabled by GEq’s features, along with related and future work.

5.2 Setting The Stage

5.2.1 Programming vs. Proving and the Gradual Guarantees

Though programming and proving are connected by the Curry-Howard correspondence, the pragmatics of proving and programming are somewhat different. Our focus is dependently typed programming: we consider GEq as a model of a programming language rather than as a type theory for mechanizing mathematics. Nevertheless, we prove important metatheory about GEq that may aid in the development of future gradual type theories.

One goal with GEq is proving the gradual guarantees of Siek et al. [136], which state that a reduction in precision introduces no new static or dynamic errors. These guarantees are useful for programming because of the contrapositive: if a
program has a type error, adding more type information does not remove the error. The types are fundamentally inconsistent and must be changed. By contrast, current implementations of holes either block reduction, causing errors, or block type checking, hiding errors that would otherwise be statically detectable.

5.2.2 Relationship to Existing Languages

GEq builds on the languages of Chapters 3 and 4, gdtl [58] and gcic [87]. Section 5.6.3 gives a broader discussion of related work.

From Gradual Dependently Typed Language (gdtl), GEq inherits ? and the general model of imprecise terms and types. Since it is based on agt [8, 67], gdtl features ideas similar to GEq’s witnesses. However, the original presentation only discuss equality and inductive types as an extension, omitting it from their metatheory. Also, gdtl uses naive syntactic composition, and suffers from the issues we discuss in the introduction: fully static terms that are observationally equivalent in the static language may have different run-time behaviour in the gradual language.

Unlike gdtl, gcic uses a cast calculus approach, extending a restricted version of cic with inductive types but no indexed inductive families or propositional equality. GEq is a direct extension of gcic. The gcic authors prove that no gradual language can simultaneously conservatively extend cic, have strong normalization, and have graduality, a strengthening of the gradual guarantees where decreasing then increasing precision produces an equivalent term. The authors give three variants of gcic, called gcic^G, gcic^N and gcic^\textdagger, which respectively sacrifice one of strong normalization, graduality, and conservative extension of cic, while keeping the other two properties. We extend gcic^G because, of the three options, we feel that sacrificing strong normalization is most palatable for programming. gcic^N violates the gradual guarantees, and gcic^\textdagger is too restrictive for practical programming, so we avoid them both. Logical inconsistency and possibly-diverging proofs are not as detrimental in programming as in mechanized mathematics: type safety is still guaranteed, and errors due to diverging proofs are likely to be discovered when a program is run.

Gcic has no dedicated equality type, but we should be clear that reasoning
about equality is not totally unsupported. Programmers can write their own *decidable equality*: a function that takes two elements of a type, and produces a type that is inhabited if and only if they are equal. The programmer must construct, either manually or with tactics, an equality function for each type whose terms they wish to equate, along with the corresponding elimination principle. Most function types have undecidable equality, and hence are unsupported by this method. Also, common functions on equalities cannot be expressed in their most general form with this method, such as $\text{cong} : (f : T \to S) \to a =_T b \to f\ a =_S f\ b$.

Full propositional equality is more convenient for the programmer.

### 5.2.3 A Motivating Example: Eagerly Enforcing Specifications

In this section, we motivate our development with examples of how gradual dependent types can catch errors related to the lengths of lists. A guiding principle of this chapter is that, to the extent possible, the types the programmer writes should be treated as specifications to be checked, either statically or dynamically. These checks should happen regardless of whether their enforcement is required for safety. We revisit the quicksort example of Section 1.3.1.1, but instead of vectors, use the equivalent type of fixed-length lists. Though the notation is more cumbersome, the advantage is that, unlike Section 3.2, we no longer need to hand-wave over the lack of support for indexed families. The terms in the following examples can actually be written using the features of GEq.

#### 5.2.3.1 A Buggy Quicksort

Once again we start with quicksort, but this time, we begin with code containing a mistake, to show how gradual dependent types can possibly help the programmer find bugs:

```plaintext
sort : List Float → List Float
sort Nil = Nil
sort (Cons h t) =
  (sort (filter (< h) t)) ++ [h] ++ (sort (filter (> h) t))
```

134
Since $<$ is used instead of $\leq$, duplicates are erroneously removed from the list. The programmer may have a suspicion that they have made a mistake in their code, or may have observed incorrect behaviour while testing. Their dependent type enthusiast friends have repeatedly assured them that dependently typed languages can help eliminate bugs, so they try migrating their code to a dependently typed language with propositional equality.

Just like we used vectors in Section 1.3.1.1, an approach to producing a bug-free quicksort is to use fixed-length lists, which we call FLists. Dependent pairs and propositional equality allow for a type of lists indexed by their length. Fixed length lists are the dependent-pair version of vectors from Section 1.3.1.1, which we avoid in order to stay strictly within the features of GEq. Full inductive families are discussed in Section 5.6.1.

Figure 5.1 shows the definition of FList, and how it can be used to express that sort should preserve the length of the produced list. Here, refl is the reflexive proof that $t$ is equal to itself. That sort function behaves like the non-dependent version, except it must extract the Lists from the FLists produced by the recursive calls, and produce an FList with proof that the length is the same as the input.

At this stage, the programmer must fill the hole ???? by constructing a proof of type:

$$\text{length (Cons } h \ t) =_{\mathbb{N}} \text{length (sortLt + [h] + sortGt)}.$$  

The programmer must prove that $\text{length } t =_{\mathbb{N}} (\text{length } lt) + (\text{length } gt)$, but such a proof is impossible, due to the bug. Even if they had such a proof, they would need to then use the proofs from the recursive calls, plt and pgt, to relate the lengths of lt and gt to the lengths of sortLt and sortGt.

Development has now stopped: the programmer cannot run or test their code without the missing proof, and they are now cursing their type-theorist friend. For a non-buggy quicksort, one could construct the necessary proof, but doing so is difficult, particularly for a newcomer. The type checker does not detect the bug, so it does not inform the programmer that hole cannot be filled, and it cannot say which aspect of the proof is impossible.
FList : Type → ℕ → Type
FList A n = ((x : List A) × length x =ₙ n)

sort : (n : ℕ) → FList Float n → FList Float n
sort 0 (Nil, p) = (Nil, p)
sort (1 + n) (Cons h t, p) =
  let lt = (filter (< h) t)
      (sortLt, plt) = sort (length lt) (lt, refl)
    gt = (filter (> h) t)
      (sortGt, pgt) = sort (length gt) (gt, refl)
in (sortLt + [h] + sortGt, ???)

Figure 5.1: Sorting Fixed-Length Lists

5.2.3.2 Gradual Types to the Rescue

GEq lets the programmer run and test sort before writing the missing proof, checking (within the limits of decidability) whether any static values could possibly replace imprecise types and proofs. In Fig. 5.1, replacing the hole with ? , the imprecise term, \(^1\) yields a complete, well typed GEq function that can be called, tested, or used in other modules.

The bug-finding potential of gradual propositional equality is shown when we actually run the code. Like gcic, the run-time semantics of GEq are defined via type-directed elaboration to a Cast calculus with propositional Equality (CASTEQ), in which all implicit conversions are replaced by explicit casts. During type checking, ? is elaborated into the CASTEQ term:

\[ ?_n=\langle \text{length} \ (\text{sortLt}+ [h] + \text{sortGt}) \]\n
i.e., the least precise term of type \( n =\langle \text{length} \ (\text{sortLt}+ [h] + \text{sortGt}) \), which is

\(^1\)Each ? is actually \( ?_{\leq 0} \), i.e., annotated with its type’s universe level. Our exposition omits levels; we explain them in Section 4.4.
not a value in \texttt{CastEq}. Instead, it reduces to the consistency witness

\[
n \&_N \text{length} (\text{sortLt} + [h] + \text{sortGt})
\]

The operator \& is the gradual composition operator, a first-class version of composition \& from \texttt{GDTL}, which combines information statically known about its operands. Because types depend on terms, composition is not limited to types, but can combine terms of any type. The \& operator is a syntactic construct of \texttt{CastEq}, not a meta-operation like it is in existing literature [130, 135]. Reifying \& into the object language is critical for composing functions (Section 5.2.4). Since \(n\) is a variable, this composition expression does not reduce further.

We can identify the bug once \texttt{sort} is applied to a concrete list. Consider the input \([2.2, 1.1, 3.3, 2.2]\), which elaborates to \([2.2, 1.1, 3.3, 2.2]\) in \texttt{CastEq}. Applying \texttt{sort} binds \(\texttt{lt} := [1.1]\) and \(\texttt{gt} := [3.3]\), giving a result list of \([1.1, 2.2, 3.3]\). Then \(n\) is 4 and \texttt{length (sortLt + [h] + sortGt)} is 3, so the witness for the result is the composition \(4 \&_N 3\), which reduces to a run-time error.

In a language without dependent types, this bug could be caught with testing or assertions. In \texttt{GEq}, however, dependent types provide a unified means of specifying properties to be checked statically or dynamically. During development, types serve as assertions to be checked dynamically (or statically, if enough information is present). When a program is completed and all uses of \texttt{?} have been removed, those same types establish properties that have been statically verified.

### 5.2.3.3 Witness Composition

The key to finding the error above was tracking information with witnesses, and combining those witnesses using the composition operator. While that composition was a simple equality check, in general the composed values may be imprecise, and the result is some value that is as precise as both inputs. The information from the witness is used when eliminating an equality proof: when using a witness \(t_w\) of \(t_1 \equiv T t_2\) to rewrite a term of type \(P(t_1)\) into \(P(t_2)\), we first cast to \(P(t_w)\), then to \(P(t_2)\). For a program with imprecise types or values, the witness retains the information gained by running the program, preventing unsafe execution, and informing the programmer when a counter-example to an imprecise equality is
Here we present an example of a bug that is found, not because of safety, but because a remembered constraint was violated. Consider the following functions:

\[
\text{zip} : (n : \mathbb{N}) \rightarrow \text{FList} A n \rightarrow \text{FList} A n \rightarrow \text{FList} (A \times A) n
\]
\[
\text{take} : (n : \mathbb{N}) \rightarrow (m : \mathbb{N}) \rightarrow \text{FList} B (n + m) \rightarrow \text{FList} B n
\]

Here, \textit{zip} takes two lists of exactly the same length, and produces a list of pairs of their elements, while \textit{take} takes a list with at least \(n\) elements, and returns the first \(n\) elements of that list. Each function constrains the size of its input, so by tracking equality witnesses, we can also track these constraints and detect where they are incompatible. Now consider lists with imprecise types:

\[
\text{list}_1 := ([1.1, 2.2], \text{refl}) : \text{FList} \text{ Float} ?
\]
\[
\text{list}_2 := (\text{Cons} 1.1 ?, \text{refl}) : \text{FList} \text{ Float} ?
\]

For \text{list}_1, we are converting a list of length 2 to a fixed-length list of unknown length, since 2 is consistent with \(?\). For \text{list}_2, however, the length is truly imprecise, since its tail is the unknown term. We can zip these lists together as

\[
\text{zip} ? \text{list}_1 \text{list}_2 : \text{FList} \text{ Float} ?
\]

producing another list of unknown length, since recursively applying \text{zip} to the unknown tail \(?\) produces an unknown result. Applying \text{take} 3 to the result of \text{zip} is well typed, since the length \(?\) is consistent with \(3 + ?\), i.e.,

\[
\text{take} 3 ? (\text{zip} ? \text{list}_1 \text{list}_2) : \text{FList} \text{ Float} 3
\]

However, computing the witness flags an error.

This error represents something deeper than a simple safety check: it detects fundamental inconsistencies in statically-determined propositional equalities. In the absence of equality proofs, the call could run safely: \textit{Cons} (1.1, 3.3) ? would be a sensible result, having length consistent with 3. The witness composition is not just checking if a list is empty before taking the head, or counting the elements
in the list before running `take`. Rather, the information added by `zip`, that the list should have length 2, has been propagated using the `list₁` witness and composed with the conflicting information. GEQ uses witnesses to enforce imprecise equality constraints at run time.

To understand how GEQ detects this mismatch, we look at the result of elaborating to `CastEq`. Initially, `list₂` has `1 + ?ₙ₁` as the witness that `?ₙ₁` is equal to `1 + ?ₙ₁`. The result of `zip` has `2` as the witness of equality between `?ₙ₁` and `?ₙ₁`, since that is the length of `list₁`. This new witness was determined by composition: since `1 + ?ₙ₁` is consistent with `2`, this composition succeeds. (Using a Peano representation of naturals, `S(?)` is consistent with `S(S(0)))` Then, even though `zip`’s result has a type that is consistent with what `take` expects, the run-time type information remembers that `zip` constrained the list to have length `2`. The result of `zip` is cast to `FList Float (3 + ?ₙ₁)`, the type expected by `take`. The `zip` result has an equality proof of type `1 + ?ₙ₁ =ₙ ?ₙ₁`, which is cast to type `1 + ?ₙ₁ =ₙ 3 + ?ₙ₁`. During this cast, the target value `3 + ?ₙ₁` is composed with the witness `2`. Despite the imprecision, these values are not consistent, and composition produces an error: no value can replace `?` to make `S(S(S( )))` equal to `S(S(0))`. We detail the semantics enabling this in Section 5.3.

With equality witnesses, we achieve more than type safety. From the gradual guarantees, we know the above code cannot possibly be made static by replacing the `?` uses with static terms. When a witness reduces to an error, the program is equating two terms that are fundamentally not-equal. So the gradual guarantees now inform about equality constraints, in addition to type constraints. These constraints are expressed through types, rather than some external language of assertions.

5.2.4 Lazily Enforcing Specs: Function Equalities and Extensionality

Propositional equality is not restricted to first-order values like numbers or to types with decidable equality. In particular, we can form equalities between functions, for which equality is not in general decidable. The following summarizes how GEQ handles propositional equality for functions without encountering the
impossibility result of Lennon-Bertrand et al. [87]. Consider the example they use to show the incompatibility between gradual typing and refl-based equality:

\[
\begin{align*}
\text{id}_\mathbb{N} &:= (\lambda x. x) : \mathbb{N} \rightarrow \mathbb{N} \\
\text{add}0 &:= (\lambda x. x + 0) : \mathbb{N} \rightarrow \mathbb{N} \\
\text{test} &:= \lambda f. J \ (\_ \ _ \ _ \ B) \ \text{id}_\mathbb{N} \ \text{true} \ f \ (\text{refl}_{\text{id}_\mathbb{N}} :: ? :: (\text{id}_\mathbb{N} =_{\mathbb{N} \rightarrow \mathbb{N}} f)) : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow B
\end{align*}
\]

Here \( J \) is the eliminator for equality: we explain it fully in Section 5.3.2, but it suffices to know that in this case, it uses a proof of type \( \text{id}_\mathbb{N} =_{\mathbb{N} \rightarrow \mathbb{N}} f \) to rewrite \( (\lambda x. x) \ \text{id}_\mathbb{N} \) to \( (\lambda x. x) \ f \). The form :: denotes surface type ascriptions, which are elaborated to casts in the cast calculus. Both types reduce to \( B \), but \( J \) only reduces if the equality proof reduces without error, which causes problems with test.

Lennon-Bertrand et al. [87] show that test distinguishes \( \text{id}_\mathbb{N} \) and \( \text{add}0 \). When \( \text{id}_\mathbb{N} \) is given as an argument, casting \( \text{refl}_{\text{id}_\mathbb{N}} \) to ? then back to \( \text{id}_\mathbb{N} =_{\mathbb{N} \rightarrow \mathbb{N}} \text{id}_\mathbb{N} \) should produce \( \text{refl}_{\text{id}_\mathbb{N}} \). However, when \( \text{add}0 \) is given for \( f \), the cast must fail, since \( \text{refl} \) cannot have type \( \text{id}_\mathbb{N} =_{\mathbb{N} \rightarrow \mathbb{N}} \text{add}0 \).

This example captures the subtle way that gradual typing can observe information that is transparent to static programs. Since \( \text{id}_\mathbb{N} \) and \( \text{add}0 \) agree on all inputs, they should be observationally equivalent, producing the same result in any context in which we use them. Violating this would mean that the embedding of cic into gfcic or GEQ does not respect function extensionality, i.e., some statically-equivalent terms are distinguishable in the gradual language. Even though there are dependent type theories in which extensionally equal functions are not provably equivalent, the problem is that the gradual language has destroyed the ability to mentally reason about equivalent pieces of code using principles from the static language.

The first key to ensuring such terms remain equivalent is the witness-based representation of equality. In CastEq, \( \text{refl} \) is the only constructor for equality, but it takes an argument: the consistency witness for the equated terms. Moreover, it does not require the equated terms to be syntactically identical, only that the witness be at least as precise as both of them. So \( \text{refl} \) can have type \( \text{id}_\mathbb{N} =_{\mathbb{N} \rightarrow \mathbb{N}} \text{add}0 \). But what witness should be attached to the proof of \( \text{id}_\mathbb{N} =_{\mathbb{N} \rightarrow \mathbb{N}} \text{add}0 \)?

The second key feature is the composition operator of CastEq, which is used
to build equality witnesses. Elaborating $\text{refl}_{id_N}$ creates witness $id_{N!}$. The cast to $id_{N!} =_{N \to N} f$ composes that witness with the destination endpoints, $id_{N!}$ and $f$, yielding $id_{N!} &_{N \to N} id_{N!} &_{N \to N} f$. The semantics of $\&$ reduce this composition to $\lambda x. (x &_{N!} x &_{N!} (f x))$, similar to how a higher-order contract applied to a function produces a new function that checks the input and result [65]. Since $\&$ is an operator in the language, the composition does not need to reduce further, but when the function is applied, it continues to reduce. The same holds when we replace $f$ with $id_N$ or $\text{add} 0$. Section 5.5 defines precision such that $x &_{N!} x &_{N!} f x$ is more precise than both $x$ and $f x$, so the above composition is a valid witness. We define semantics for $J$ so that when it is given the equality proof with the above witness, it reduces, so $\text{test}$ reduces to $\text{true}$ for both $id_{N!}$ and $\text{add} 0$.

How can equating non-identical functions be safe, since deferring composition means that we can prove an equality between unequal functions? As we saw with $\text{sort}$ above, $J$ casts through the witness, so when functions are extensionally non-equal, trying to prove equality between their results dynamically fails. Consider instead $\text{subadd1} := (\lambda (x : N). x - 1 + 1)$. Since $0 - t = 0$ in $N$, $\text{subadd1} 0 = 1$. We can use the witness $\lambda x. (x &_{N!} x &_{N!} (\text{subadd1} x))$ to inhabit $id_{N!} =_{N \to N} \text{subadd1}$. However, if we try to use $J$ to prove that $id_{N!} 0 =_{N!} \text{subadd1} 0$, the result substitutes $0$ for $x$ in the witness, giving $0 &_{N!} 0 &_{N!} 1$, which reduces to an error.

The consequence of our approach is that GEQ supports a limited form of extensionality. Neutral terms, i.e., variables or terms for which reduction is blocked by one or more variables, always compose to a non-error, so we can build a witness capturing the plausibility of equality between them, given partial information. That witness makes an equality proof constructible. Furthermore, any two functions with neutral bodies compose to a non-error. If the functions agree on all inputs, eliminating their equality never fails and the proof of equality can be freely used. If the functions disagree on some input, an error is raised when building a term that relies on the functions producing the same value for said input. Since it is undecidable whether two functions agree on every input, this approach finds a balance between decidability and flexibility.
5.2.4.1 Static vs. Dynamic Consistency

The relationship between consistency and composition is complicated by dependen-
dent types. For non-dependent gradual types, the successful composition of two
types usually implies that they are consistent. However, for GEQ, two neutrals al-
ways compose to a more-precise term. To conservatively extend Cic, all ill typed
(fully-static) Cic programs must be ill typed in GEQ, and an uninhabited Cic type
must not be inhabited by any fully-static GEQ terms. So GEQ cannot have all neu-
trals consistent: this would yield a fully-static proof of \((\lambda x. \lambda y. x) =_{T \rightarrow T} (\lambda x. \lambda y. y)\).

We resolve this tension with separate static and dynamic notions of consist-
cy (Section 5.5). Terms are statically consistent if they are syntactically equal
up to \(\alpha\)-equivalence, reduction, casts, and occurrences of \(?_T\). Terms are dynam-
ically consistent if they compose without error, or equivalently, if there exists
a non-error term as precise as both terms. Essentially, terms are dynamically-
consistent if they are statically consistent in the non-neutral parts. The type rules
for GEQ use static consistency. Some pairs of terms are not statically consistent,
yet still compose to a non-error term.

To compare static and dynamic consistency, consider the ill typed Cic term
\texttt{refl : x =_N y}. When embedded into GEQ, \texttt{refl : x =_N y} is still ill typed: the expected
type of \texttt{x =_N y} and the actual type of \(x =_N x\) are not statically consistent, because
the variables \(x\) and \(y\) are not identical. In CastEq, \(x\) and \(y\) are neutral, and hence
dynamically consistent, meaning \(x \&_N y\) witnesses \(x =_N y\).

Letting neutrals be dynamically consistent does not interfere with conser-
vatively extending Cic. For conservative extension, every ill typed Cic program
should be ill typed in GEQ. In the absence of \(?\), pairs of definitionally-unequal
Cic terms are statically inconsistent. While GEQ gives \texttt{refl} the same type as Cic,
CastEq lets \texttt{refl} prove equality for dynamically consistent terms. However, dy-
namic consistency does not let CastEq type any ill typed Cic terms, because Cic
programs are elaborated into a subset of CastEq where \(t\) only witnesses \(t =_T t\)
and all casts have the form \((T \leftarrow T)\). The type \((x : \_N) \rightarrow (y : \_N) \rightarrow (x =_N y)\) is un-
inhabited in Cic, and while this type is inhabited in CastEq using witness \(x \&_N y\),
the use of \(\&_N\) puts the witness outside the static fragment of CastEq. The term
\texttt{x \&_N y} does not correspond to any typed or ill typed Cic program.
Static and dynamic consistency let us balance conflicting goals. If all statically inconsistent functions composed to an error, then statically-equivalent terms would not be gradually equivalent, making it harder to reason about program equivalence. Using dynamic consistency during type checking would not conservatively extend cic. By separating these, we obtain conservative extension, while dynamically respecting all extensional equalities.

5.3 Propositional Equality

The main contribution of this chapter is GEQ: an extension of gcic with propositional equality, where the information about an imprecise value accumulates at run time to detect inconsistencies. We define GEQ’s semantics using a cast calculus CASTEQ, which extends Cast calculus for the Gradual Calculus of Inductive Constructions (CASTCIC) with equality.

The core idea is that a surface-language proof of type \( t_1 =_{T} t_2 \) is elaborated into a witness for the consistency of \( t_1 \) and \( t_2 \). Much like evidence\(^2\) from AGT [67] or the middle-type from threesomes [135], the consistency witness between terms is a term that is at least as precise as either term. The standard equality proof, \( \text{refl} : t =_{T} t \), witnesses that \( t \) is consistent with itself, while the imprecise proof \( ?_{t_1 =_{T} t_2} \) is witnessed by the least precise term that is dynamically consistent with \( t_1 \) and \( t_2 \). As a program runs, equality witnesses may take values between these extremes, which may be more precise than the witness for \( \text{refl} \) when \( t \) is imprecise.

The technical challenge with adding propositional equality is determining how to combine information represented by the equality witnesses. When casting between types \( t_1 =_{T} t_2 \) to \( t_0' =_{T} t_0'' \), both of which may be imprecise, we must transform a witness \( t_w \) for \( t_1 =_{T} t_2 \) into one for \( t_1' =_{T} t_2' \), but even though \( t_w \) is as precise as \( t_1 \) and \( t_2 \), it may not be as precise as \( t_1' \) and \( t_2' \). So we need a composition operator that can take \( t_1' \), \( t_2' \) and \( t_w \) and produce a term that is as precise as all three. However, to respect static observational equivalences and avoid the problems of Section 5.2.4, composition cannot be a syntactic meta-operation. The issue is with neutral terms, i.e., variables, or terms whose reduction is blocked by

\(^2\)Evidence is more complex in AGT, since it can witness subtyping. Evidence for plausible equality between types collapses to a single term as precise as the equated terms.
applying or eliminating a variable. Syntactic composition would require distinct neutral terms to compose to an error, but that would violate static equivalences.

Along with composition, we must define a notion of precision that determines valid witnesses of consistency. For the evolution of type information to be monotone, the operator & should compute a lower bound with respect to this notion of precision. Computing the greatest lower bound prevents premature errors, although the proof that composition is the greatest lower bound is left to future work. With non-dependent gradual types, precision can be defined syntactically, by adding structural rules to \( t \subseteq \tau \), but structural rules are not flexible enough to handle composition.

The solutions to these two challenges are interdependent. We avoid the issues with syntactic composition by adding it as a separate syntactic construct to CastEq, so that composition of neutral terms is itself a neutral term. However, if composition is a construct in CastEq, then precision must be defined to accommodate terms that feature composition, so composing two neutral terms produces something that is actually as precise as those two terms. Precision must be defined to respect composition without losing its other important properties, such as transitivity.

This section gives typing and semantics for gradual propositional equality, where proofs of equality are represented by consistency witnesses, but at this point we leave the exact definitions of consistency and precision unspecified. In Section 5.4, we describe the properties that consistency and precision should fulfill to ensure that GEQ satisfies type safety and the gradual guarantees. Finally, Section 5.5 instantiates GEQ with notions of precision and consistency that fulfill our goals while ensuring decidable consistency-checking. We separate our presentation in this way to motivate the choices we make in the design of precision, and to avoid monolithic proofs when developing GEQ’s metatheory.

We write precision as \( \Gamma \downarrow \Gamma \vdash t_1 \equiv t_2 \) and consistency as \( t_1 \equiv t_2 \), highlighting the operators in grey to indicate that their definitions are not yet specified. The subscript \( \equiv \downarrow \) on \( \equiv \) indicates that it is definitionality consistency, whose name is chosen by analogy to definitionality equality, since the operands can be reduced before being compared structurally [92]. Precision is precision modulo conversion, meaning it is closed under the equivalence relation given by convertibility. Unlike
consistency, precision modulo conversion can look backwards in time, relating terms that are the results of reducing syntactically-related terms, in addition to relating terms that are syntactically-related after reducing. Thankfully, this time-travelling relation is only needed for the theory, not to type check or elaborate gcic programs. We discuss the need for this in Section 5.3.2. Precision takes two contexts, as its operands must be typed in different contexts.

5.3.1 GEq Syntax and Typing

Figure 5.2 extends gcic to GEq by adding the equality type, introduction form, and eliminator. Their types are identical to what is expected in the static setting. Again, because surface typing is defined by elaboration, the given rules are actually admissible lemmas. An equality type \( t_1 =_T t_2 \) denotes equality between any two values of consistent types (because each endpoint is checked against \( T \)). The reflexive proof \( \text{refl}_t \) synthesizes type \( t =_T t \), so long as \( t \) is well typed at type \( T \). The eliminator \( \text{J} \) takes a type \( T_P \) parameterized over a value of type \( T^3 \), along with two values of type \( T \). Then, given a value of type \( [t_1/x] T_P \), and a proof \( t_{eq} \) that \( t_1 \) and \( t_2 \) are equal, the elimination has type \( [t_2/x] T_P \). That is, if two values are equal, we can take any term whose type refers to the first, and transform it into a term whose type refers to the second. For the remainder of this dissertation, we use the notation \( [t/x]u \) to denote the usual capture-avoiding substitution, not the hereditary substitution of Chapter 3.

5.3.2 CastEq Syntax and Typing

Figure 5.3 extends the CastCIC to CastEq by adding propositional equality and the gradual composition operator. We extend the syntax for a static head \( h \) to include value constructors, not just types, which is useful when defining the semantics of composition. A proof of reflexivity is written as \( \text{refl}_w((t_w)_{t_1=t_2}) \), where \( t_1 \) and \( t_2 \) are the equated terms, and \( t_w \) is a witness of the (dynamic) consistency of those endpoints. We borrow the notation \( (t_w)_{t_1=t_2} \) from agt to indicate that \( t_w \) contains information supporting the (dynamic) consistency of \( t_1 \) and \( t_2 \). Compo-

\[ \text{The full J in type theory parameterizes } T_P \text{ over the equality proof. Section 5.6.1.2 shows why this is not needed for GEq.} \]
\[
\begin{align*}
\Gamma \vdash t_1 \Leftarrow T & \quad \Gamma \vdash t_2 \Leftarrow T & \quad \Gamma \vdash \text{Type} \quad \Gamma \vdash t_1 \Rightarrow \text{Type} \quad \Gamma \vdash t_1 \Rightarrow \text{Type} \\
\vdash t + : = t = _T t_1 | \text{refl} | J(x, T, t_1, t_2, t_3, t_4) \\
\vdash h + : = \\
\end{align*}
\]

Each new form also has an associated typing rule. \textit{CASTCOMP} synthesizes a composition’s ascribed type when both arguments check against that type. In \textit{CASTREFL}, \texttt{refl(t_w), t_1 \Rightarrow t_2} synthesizes \( t_1 \Rightarrow t_2 \) if the witness \( t_w \) is as precise as both \( t_1 \) and \( t_2 \). In \textit{ELABREFL}, \texttt{refl} is elaborated into \texttt{refl(t), t=t}, i.e., a term serves as the initial witness that it is equal to itself. If \( t \) is imprecise, casts applied to the equality proof may produce more precise witnesses, but the programmer never constructs a witness directly.

The typing rule for \textit{CASTREFL} induces constraints on our precision relation, which we note here so we can address them when fully defining precision. Precision must be closed under convertibility because, as Lennon-Bertrand et al. [87] note, syntactic precision is not preserved by stepping the less precise term. Since \( ?_T \) is less precise than \( x \), the less precise term may reduce in a way that is blocked for the other term. So for contextual steps to preserve \textit{CASTREFL}, the results of stepping related terms must be related.

5.3.3 Cast Semantics

Much of the challenge with gradual equality is designing its dynamic semantics. In a fully static language, \texttt{refl} always equates identical values, so \texttt{J} performs no computation other than pattern matching on the proof of equality. In the presence
Figure 5.3: CASTEQ: Syntax, Typing and Elaboration Rules
of type imprecision, \( J \) must perform casts. \( \text{RedEqGerm} \) casts an equality proof to \( ?\text{Type} \), by casting through the germ type, just like with functions and constructors.

We also need reductions for casts between equality types. Figure 5.4 gives reductions for \( J \) and casts. The \( \text{RedJ} \) rule reduces by casting through the motive \( T_P \) with \( x \) bound to the witness \( t_w \). The typing of equality guarantees that this witness \( t_w \) is as precise as either \( t_1 \) and \( t_2 \). So \( [t_w/x]T_P \) is like a middle type, since it is more precise than \( [t_1/x]T_P \) (the type of \( t_{P1} \)) and \( [t_2/x]T_P \) (the type of the result).

Why cast through the middle, and not directly from \( [t_1/x]T_P \) to \( [t_2/x]T_P \)? As Section 5.2.3 showed, the witness tracks constraints as the program runs, and since composition is monotone, its precision only increases. So constraints are remembered, and \( J \) only succeeds if the equated terms are consistent with all those remembered constraints, letting the programmer see when a static constraint has been dynamically violated. Also, the witness ensures that equalities between inconsistent values cannot be used without flagging an error. Without a witness, one could have \( ?2 = \text{Nat} \), despite the type being statically uninhabited. Then \( J \) could use this equality to convert from \( \text{Vec Float (2 mod 3)} \) to \( \text{Vec Float (5 mod 3)} \); the cast would succeed, despite the absurdity of the initial equality. Going through
the middle type catches such absurd cases.

In addition to eliminating equality, we must also be careful when defining semantics for the casts between equality types. The REDCASTEQ rule first casts the witness to the correct type. The typing rule CASTREFL requires the witness to be as precise as the endpoints, but the result of casting the witness might not fulfill this! So the witness is composed with both endpoints, producing a precision-related result. These casts are precisely why we need a composition operator.

A key feature of GEQ is that equality is, in some ways, treated as a negative type. In particular, just like functions in GCIIC, there is no separate construct for ? or _U_. Instead, the propagation rules PROP_EQUNK and PROP_EQ_ERR reduce ? and _U_ at equality types to refl with the least and most precise witnesses, respectively.

5.3.4 Semantics of Composition

In Fig. 5.5, we define the semantics for composing arbitrary CASTEQ terms. Technically, we do not need composition as an operator in CASTEQ itself, but only for the subset of GEQ used in witnesses and cast type ascriptions. However, because dependent types remove the separation between terms and types, witnesses and cast types need dynamic semantics. So for simplicity, we let witnesses and cast types be any CASTEQ terms, and add composition to CASTEQ’s semantics, rather than duplicating CASTEQ’s semantics for a witness-specific language.

Several composition rules receive two or three different sets of type ascriptions, with one on the composition itself, and possibly one on each of the composed terms. In such cases, the choice of ascription in the reduct is arbitrary, because typing guarantees what value it should have, modulo conversion. For example, REDCOMPEQL uses the ascription from _U_, rather than what was on &. While we do not require that the ascriptions be syntactically equal, the typing rules ensure that syntactically distinct ascriptions are definitionally equal, so the choice of ascription in the reduct does not affect the final result of evaluation.

Each composition rule resembles a unification rule, where each use of ? is treated as a unification variable. Terms that have different head tags compose to an error, and terms that have the same head tag compose by composing their parts. Any place where one term has ?, but the other term contains more pre-
\[
\begin{align*}
\text{RedCompUnkL} & \quad \text{UnkVal } T \\
\text{RedCompUnkR} & \quad \text{UnkVal } T \\
\text{RedCompErrL} & \quad \text{UnkVal } T \\
\text{RedCompErrR} & \quad \text{UnkVal } T \\
\text{RedCompGerm} & \\
\left(\begin{array}{l}
\text{Type}_r \leftarrow \text{germ}(h) t_1 \\
\text{Type}_r \leftarrow \text{germ}(h) t_2
\end{array} \right) & \sim \\
\left(\begin{array}{l}
\text{Type}_r \leftarrow \text{germ}(h)(t_1 \& \text{germ}(h) t_2)
\end{array} \right)
\end{align*}
\]

\[
\begin{align*}
\text{RedCompGermErr} & \\
(t_1 \neq t_2) & \sim U_{T_{\text{Type}_r}}
\end{align*}
\]

\[
\begin{align*}
\text{RedCompHeadErr} & \\
\text{head}(t_1) = h_1 & \quad \text{head}(t_2) = h_2 \\
(h_1 \neq h_2) & \sim U_T
\end{align*}
\]

\[
\begin{align*}
\text{RedCompEq} & \\
(t_1'' : (T_1, \text{Type}_r, T_2 \leftarrow T_1, T_2 \leftarrow T_1' & \leftarrow T_2) & \sim \left(\begin{array}{l}
T_1 & \leftarrow T_1', T_2 & \leftarrow T_2'
\end{array} \right)
\end{align*}
\]

\[
\begin{align*}
\text{RedCompRefl} & \\
\text{refl}(t_1) & \sim \text{refl}(t_1)
\end{align*}
\]

\[
\begin{align*}
\text{RedCompCon} & \\
\text{D}(F_1, F_2) & \sim \text{D}(F_1, F_2)
\end{align*}
\]

\[
\begin{align*}
\text{RedCompInd} & \\
\text{C}(T_1) & \sim \text{C}(T_1)
\end{align*}
\]

\[
\begin{align*}
\text{RedCompPi} & \\
(x : T_1) & \rightarrow T_1' & \& \text{Type}_r, (x : T_2) & \rightarrow T_2' & \sim \\
(x : \left(\begin{array}{l}
T_1 & \leftarrow T_1', T_2 & \leftarrow T_2'
\end{array} \right) & \rightarrow \left(\begin{array}{l}
T_1 & \leftarrow T_1', T_2 & \leftarrow T_2'
\end{array} \right)
\end{align*}
\]

Figure 5.5: CASTEq: Semantics for Composition
cise information, the precise information is retained in the output. We only apply this rule for positive types (defined in Fig. 5.6) for which ? and U are values. Note that we consider equality a negative type, which is one of the insights that makes GEq work. The positivity restriction ensures that when $t_1 \&_T t_2$ reduces, the result is a term that is as precise as both $t_1$ and $t_2$. For $?_T \&_T t$, we produce $t$ (REDCompUnk(L,R)), since $t$ is always as precise as itself and $?_T$. Likewise, the rules that produce $U$ satisfy this, since it is the most precise term. We see this in REDCompUnk(L,R), which composes with $U$, and in REDCompHeadErr and REDCompGermErr, where composing non-neutral terms with distinct heads reduces to $U$.

The remaining rules compose terms with the same head $h$, such as when both are functions or both are built with the same $D_C$. In these cases, the head $h$ is applied to the respective composition of the arguments, e.g., the composition of functions is a function returning the composition of the bodies. For equality proofs and inhabitants of $?Type_r$, the head can be applied directly (REDCompLam, REDCompRefl, REDCompGerm). In REDCompLam, composing functions yields a new function that composes the results of the given functions for each argument. In the remaining cases, we must account for how types of later arguments depend on the values of earlier arguments. REDCompPi produces a domain by composing the argument domains, which is the type of the parameter $x$. The codomain $x$'s each have their own domain types, so we cast all uses of $x$ from the composed type to the expected type. REDCompEq composes equality types: the type ascriptions are composed, then the equated terms are cast to this composed type. Composing the results of these casts yields the result endpoints.

The most complex rules are REDCompInd and REDCompCon. Because type and data constructors have dependent function types, their arguments are tele-

<table>
<thead>
<tr>
<th>UnkVal T (Types with ? and U forms)</th>
</tr>
</thead>
</table>

| UnkVal Type_r | UnkVal C(?) | UnkVal ?Type_r |

**Figure 5.6: Positive Types**

151
scopes: the type of later arguments may depend on the values of previous parameters and arguments. To compose type or data constructor applications, we compose the parameters and arguments element-wise, but composing two arguments changes the type of later arguments. To compose telescopes, the metafunction \( \varphi \) traverses the types of type and data constructors, composing arguments element-wise and adding casts to the bound variables in later arguments (Fig. 5.7).

To see why composing needs casts, consider dependent pairs formulated as an inductive type:

\[
data 
  \text{DPair} : (X : \text{Type}) \rightarrow (P : X \rightarrow \text{Type}) \rightarrow \text{Type}
\]

where

\[
  \text{mkDPair} : (x : X) \rightarrow P x \rightarrow \text{DPair} X P
\]

One example of a dependent pair type is \( \text{DPair} (\mathbb{N} \times \mathbb{N}) (\lambda x. (\pi_1 x) + (\pi_2 x) =_{\mathbb{N}} 3) \), i.e., the Curry-Howard equivalent of “there exists a pair of numbers such that adding them yields 3.” Suppose we want to compose two inhabitants of this type, say \( \text{mkDPair} (1, ?_N) \) and \( \text{mkDPair} (?_N, 2) \). To compose the first element, we can produce \( (1 \& ?_N, ?_N \& 2) \), which reduces to \( (1, 2) \). However, for the second element, the two proofs \( \text{refl}(3)_{1+?_N=3} \) and \( \text{refl}(3)_{?_N+2=3} \) do not have the same type: they equate different terms, so we cannot compose them! Instead, we must first cast each to type \( (1 \& ?_N) + (?_N \& 2) =_{\mathbb{N}} 3 \), i.e., the value obtained by replacing \( x \) with the composition of the pairs’ first elements in the term \( ((\pi_1 x) + (\pi_2 x) =_{\mathbb{N}} 3) \). The final result is then \( \text{mkDPair} (1, 2) (\text{refl}(3)_{1+2=3}) \).

\[
\varphi((x : T), t_1, t_2) \quad \text{(Telescope Composition)}
\]

\[
\varphi((x : T), (y : T), t_1, t_2, t_3) := \varphi((t_1 \& T, t_2), \varphi((y : [(t_1 \& T, t_2)/x]T_2) \varphi(seq_1, seq_2))
\]

where

\[
seq_1 := \langle [(t_1 \& T, t_2)/x]T_2 \varphi[t_1/x]T_2 \varphi \rangle
\]

\[
seq_2 := \langle [(t_1 \& T, t_2)/x]T_2 \varphi[t_2/x]T_2 \varphi \rangle
\]

Figure 5.7: Telescope Composition

152
5.4 Parameterized Metatheory: Criteria for Precision and Consistency

GEQ is now defined except for CASTEq's precision and consistency relations. For non-dependent languages, the semantics of precision can be justified either in terms of sets of static terms [67] or in terms of semantic precision [105]. Such justifications are difficult with dependent types. Our approach is different: we define the important metatheoretic criteria for GEQ without referring to precision and consistency, then describe the criteria precision and consistency must fulfill to prove the desired metatheoretic properties. Meeting these criteria guides and justifies our definition of precision and consistency (Section 5.5). We see precision and consistency as a means to the end of the desired metatheory.

5.4.1 Stating the Gradual Guarantees

To state the gradual guarantees formally for GEQ, we must define what precision means for surface terms. We follow Lennon-Bertrand et al. [87] and define surface precision as the relation generated by \( t \subseteq_{\text{surf}} ?_{@i} \), plus all the usual structural rules. Essentially, \( t \subseteq_{\text{surf}} t' \) holds if we can obtain \( t' \) by replacing some parts of \( t \) with some \(?_{@i}\). To guarantee preservation of typing, we also need such replacements to be universe adequate [87]. We say that the judgment \( t \subseteq_{\text{surf}} t' \) is universe adequate if, for every subterm \( r \) of \( t \), when \( \Gamma \vdash r \rightarrow r \Rightarrow T \) and \( \Gamma \vdash T \Rightarrow \text{Type Type} \), then any uses of \( r \subseteq_{\text{surf}} ?_{@j} \) have \( i = j \). This essentially says \( t \) is \( t' \) with some subterms replaced by \(?_{@i}\) for the right \( i \). We can now state the static gradual guarantee:

**Definition 5.4.1 (Static Gradual Guarantee).** If \( \cdot \vdash t : T \) and \( t \subseteq_{\text{surf}} t' \) universe-adequately, then \( \cdot \vdash t' : T \).

That is, reducing the precision of a program causes no new type errors.

To state the dynamic guarantee without referring to CASTEq precision, we must formalize what it means to introduce no new dynamic errors. We follow New and Ahmed [105] and do this with semantic precision, which compares terms by quantifying over all possible boolean contexts. We use booleans because of their simplicity: gradual booleans have only the four values \texttt{true}, \texttt{false}, \texttt{?B} and \texttt{U_{E}}. If a context exists such that reducing a term’s precision changes the result...
from \texttt{true} to \texttt{false}, then we have violated the guarantee that precision only affects behaviour via errors. Likewise, if a context exists such that reducing precision turns \texttt{true} to $\mathbb{U}$, then reducing precision introduced a new error. By defining semantic precision in terms of \textit{all} contexts, we capture the idea that the above behaviours are impossible for precision-related terms. We formalize this as follows:

\textbf{Definition 5.4.2 (Semantic Precision).} \textit{Boolean precision $\subseteq \mathbb{B}$ is defined by:}

\begin{align*}
\texttt{true} & \subseteq \mathbb{B} \texttt{true} \\
\texttt{false} & \subseteq \mathbb{B} \texttt{false} \\
\mathbb{U} & \subseteq \mathbb{B} \, b \\
\, b & \subseteq \mathbb{B} \, ?
\end{align*}

\textit{for all} $b : \mathbb{B}$. \textit{Then two closed terms are related by semantic precision, written $\triangleright t \subseteq^+ t' : T$ if, for all} $C : T \rightarrow \mathbb{B}$, \textit{whenever} $C[t] \rightarrow^* b$, \textit{then} $C[t'] \rightarrow^* b'$ \textit{and} $b \subseteq \mathbb{B} \, b'$.

Then the gradual guarantee states that reducing a surface term’s precision causes a corresponding reduction in the semantic precision of the surface terms’ elaborations.

\textbf{Definition 5.4.3 (Dynamic Gradual Guarantee).} \textit{Suppose $\triangleright t \rightarrow t \Leftarrow T$ and $\triangleright t' \rightarrow t' \Leftarrow T$. If} $t \subseteq_{\text{Surf}} t'$ \textit{universe-adequately, then} $\Gamma \triangleright t \subseteq^+ t'$.

\subsection*{5.4.2 Necessary Properties of Precision and Consistency}

Next, we list properties that, if satisfied by the $\subseteq^+$ and $\equiv^+_<$ referred to in the CASTEq typing rules, suffice to prove type safety, conservative extension of CIC, well typedness of elaboration, and the gradual guarantees. Each criterion is accompanied by a specific case of the safety or gradual guarantee proofs that motivates its inclusion. We also include criteria that composition should satisfy. While the semantics appear in a prior section (Section 5.3.4), the criteria are new from GCIC, so we list them to highlight our contribution.

Though $\subseteq^+$ is ideal for typing witnesses, it is too lenient to express the monotonicity properties of CASTEq. In particular, if the monotonicity of reduction is phrased with $\subseteq$, then consistency must also be closed under convert-
ibility, which would make it undecidable even when its operands terminate. So we introduce a strictly stronger relation \( \sqsubseteq \), which, like \( \equiv \), only compares after reductions, and not before. This distinction is acceptable because the precision side-condition of CASTREFL is never used to prove safety or monotonicity. Rather, the side-condition ensures that witnesses always entail at least as much information as the equated terms. Including the side-condition in CASTEq’s type conveniently captures the invariant that, when \( \text{refl}_t \) is elaborated with initial witness \( t \), future witnesses are never lose the information from \( t \). Moreover, this side condition justifies having \(?_t =_t t\_2\) reduce to \( \text{refl}(t_1 \&_T t_1) =_T t_2\), since it is the least precise well typed term of type \( t_1 =_T t_2\).

### 5.4.2.1 Safety and Elaboration

For elaboration to preserve types, precision must be reflexive so the initial witness for \( \text{refl} \) is valid. For safety, progress requires that each well typed non-value can step. So composition must step for each non-value. For preservation, each reduction must preserve types, including composition reductions. If composition yields a precision lower-bound and precision is transitive, the side-conditions of CASTREFL can be preserved.

**Lemma 5.4.4 (Precision Reflexive).** If \( \Gamma \vdash t \iff T \) then \( \Gamma\mid \Gamma \vdash t \sqsubseteq t \) (For ELABREFL to produce an elaboration that satisfies the \( \sqsubseteq \) side-condition of CASTREFL).

**Lemma 5.4.5 (Composition Safety).** If \( t_1 \&_T t_2 \) is not a value and \( \Gamma \vdash t_1 \&_T t_2 \iff T \), then \( t_1 \&_T t_2 \rightarrow t_3 \) for some \( t_3 \) and \( \Gamma \vdash t_3 \iff T \) (For progress and preservation).

**Lemma 5.4.6 (Composition Confluence).** If \( t_1 \&_T t_2 \Rightarrow t_3 \) and \( t_1 \&_T t_2 \Rightarrow t_3' \) maximally, then \( t_3 \Rightarrow t_3' \), where \( \Rightarrow \) is the parallel reduction relation, standard in confluence proofs [143] (For confluence, which is needed to show that \( \beta \)-reductions preserve types).

**Lemma 5.4.7 (Composition Lower Bound).** If \( \Gamma \vdash t_1 \&_T t_2 \iff T \), then \( \Gamma\mid \Gamma \vdash t_1 \&_T t_2 \sqsubseteq t_1 \) and \( \Gamma\mid \Gamma \vdash t_1 \&_T t_2 \sqsubseteq t_2 \) (Preserving the \( \sqsubseteq \) condition of CASTREFL for reduction REDCASTEQ);
Lemma 5.4.8 (Precision Transitive). If $\Gamma_1|\Gamma_2 \vdash t_1 \equiv t_2$, and $\Gamma_2|\Gamma_3 \vdash t_2 \equiv t_3 : T$ then $\Gamma_1|\Gamma_3 \vdash t_1 \equiv t_3$ (Preserving the $\equiv$ side-condition of CastRefl for reduction REDCASTEQ);

Lemma 5.4.9 (Precision Modulo Conversion). If $\Gamma_1|\Gamma_2 \vdash t_1 \equiv t_2$, where $t_2 \rightarrow^* t'_2$, then $\Gamma_1|\Gamma_2 \vdash t_1 \equiv t'_2$ (Preservation of CastRefl under contextual reduction)

5.4.2.2 Conservativity

If GEQ is to conservatively extend cic, then a fully static program should be well typed in cic if and only if it is well typed in GEQ. For the most part, the rules only differ when $?$ is involved, but the major exception is ELABCST, which let us replace a type with any consistent type (after conversion). So for fully static terms, consistency should coincide with syntactic equality.

Lemma 5.4.10 (Static Consistency). For any static terms $t_1$ and $t_2$, let $t_1$ and $t_2$ be their embedding in CastEq. Then $t_1 \equiv t_2$ iff $t_1 \equiv t_2$, i.e., if they are statically definitionally equal (For GEQ to conservatively extend cic).

5.4.2.3 Monotonicity

The last group of properties relate to the gradual guarantees. The dynamic gradual guarantee requires that evaluating precision-related terms produces precision-related results. Because of the dependency in dependent types, proving the static guarantee relies on the proof of the dynamic guarantee: ELABCST reduces types before comparing for consistency, so precision of types before reduction should be preserved, and reducing the precision of a type should make it consistent with no fewer types. Likewise, to show the static guarantee, elaboration must be monotone in both synthesized types and elaborated terms, since dependent application uses the argument’s elaboration in the return type.

Lemma 5.4.11 (Cast Monotonicity). Suppose that $\Gamma_1|\Gamma_2 \vdash t_1 \equiv t_2$, $\Gamma_1 \vdash t_1 \Rightarrow T_1$ and $\Gamma_2 \vdash t_2 \Rightarrow T_2$ where $\Gamma_1|\Gamma_1 \vdash T_1 \equiv T'_1$ and $\Gamma_2|\Gamma_2 \vdash T_2 \equiv T'_2$. Then $\Gamma_1|\Gamma_2 \vdash \langle T'_1 \equiv T_1 \rangle t_1 \equiv \langle T'_2 \equiv T_2 \rangle t_2$ (For ELABCST to produce $\equiv_S$-related elaborations for $\equiv_S$-related inputs)
Lemma 5.4.12 (Substitution Monotone). Suppose $\Gamma_1 \vdash t_1 \xrightarrow{\tau} t_2$, where $\Gamma_1 \vdash t_1 \Rightarrow T_1$ and $\Gamma_2 \vdash t_2 \Rightarrow T_2$. If $\Gamma_1(x : T_1) \Delta_1 | \Gamma_2(x : T_2) \Delta_2 \vdash t'_1 \xrightarrow{\tau} t'_2$, then $\Gamma_1[t_1/x] \Delta_1 | \Gamma_2[t_2/x] \Delta_2 \vdash [t_1/x]t'_1 \xrightarrow{\tau} [t_2/x]t'_2$ (For ELABAPP to be monotone in the return type).

Lemma 5.4.13 (Reduction Monotone). If $\Gamma_1 \vdash t_1 \xrightarrow{\tau} t_2$ and $t_1 \xrightarrow{\ast} t'_1$, then $t_2 \xrightarrow{\ast} t'_2$ for some $t'_2$ where $\Gamma_1 \vdash t'_1 \xrightarrow{\tau} t'_2$ (For DGG, to preserve ELABCST when reducing precision, and to preserve typing under contextual reduction of $\text{refl}(t_w). t_1 \equiv t_2$).

Lemma 5.4.14 (Consistency Monotone for Precision). If $\Gamma \vdash t_1 \xrightarrow{\tau} t'_1$ and $\Gamma \vdash t_2 \xrightarrow{\tau} t'_2$, and $t_1 \equiv t_2$, then $t'_1 \equiv t'_2$ (So reducing precision of $\forall$ and $\forall'$ preserves ELABCST).

Lemma 5.4.15 (Structural Precision). $\xrightarrow{\tau}$ contains all structural rules (For homomorphic elaboration rules to produce $\xrightarrow{\tau}$-related elaboration for $\xrightarrow{\text{Surf}}$-related inputs).

5.4.3 Metatheory: Proving Safety and the Gradual Guarantees

Finally, we summarize the properties that we can prove by assuming GEQ satisfies the criteria of Section 5.4.2. The general idea is that each case in the proofs either (1) is the same as the proof for gcic [87] or (2) follows directly from one of our criteria.

5.4.3.1 Type Safety

Type safety is shown in the usual way for operational semantics, via progress and preservation [155]. Each well typed CASTEQ term is either a value, or can step to a well typed term. Confluence is necessary to prove preservation for dependent types. The idea is to follow Lennon-Bertrand et al. [87], adding $t_1 \& T t_2$ as value when $t_1$ and $t_2$ are values, neither of $t_1$ and $t_2$ is $\_? \_? T$, and $T$ is not a function type.
Lemma 5.4.16 (Confluence, Progress, Preservation and Elaboration). The following hold:

- \( \rightarrow \) is confluent.
- If \( \Gamma \vdash t \equiv T \), then \( t \) is a value or \( t \rightarrow t' \) for some \( t' \).
- If \( \Gamma \vdash t_1 \equiv T \) and \( t_1 \rightarrow t_2 \) then \( \Gamma \vdash t_2 \equiv T \).
- If \( \Gamma \vdash t \rightarrow t \equiv T \), then \( \Gamma \vdash t \equiv T \).

See proof in Appendix A.

These together yield the main safety theorem.

Theorem 5.4.17 (Type Safety). If \( \cdot \vdash t : T \), then \( t \) has an elaboration that either steps to a normal form or steps indefinitely.

As a corollary, we can perform inversion on the typing derivations to obtain weak canonicity. That is, every well typed closed term that terminates steps to a canonical term of its type.

Corollary 5.4.18 (Weak Canonicity). Suppose \( \cdot \vdash t : V \). Then either \( t \) diverges, or \( t \rightarrow^* v \) where \( v \) is \( ?_V \) or \( \mathcal{U}_V \), or the following hold:

- If \( V \) is \( \lambda x. t' \) then \( v \) is \( \lambda x. t' \).
- If \( V \) is \( C \circ (i)(t_1) \) then \( v \) is \( D \circ (i)(t_2) \) for some \( D \).
- If \( V \) is \( t_1 =_T t_2 \) then \( v \) is \( \text{refl}(t')_{t_1 = t_2} \).
- If \( V \) is \( \text{Type}_1 \) then \( v \) is one of \( C \circ (i)(\bar{t}_1), (x : T_1) \rightarrow T_2, \text{Type}_{i-1} \) or \( t_1 =_T t_2 \).

See proof in Appendix A.

5.4.3.2 Conservatively Extending \texttt{cic}

Each \texttt{cic} rule has a direct analogue in \texttt{CASTEq}, so it is clear that it extends \texttt{cic}. Since most of the gradual-specific rules refer to \( ? \) or \( \mathcal{U} \), knowing that consistency collapses to \( \alpha \)-equivalence on static terms is enough to show that said extension is conservative.
**Theorem 5.4.19 (Conservativity).** For any the Bidirectional Calculus of Inductive Constructions (bcic)-terms \( t \) and \( T \), let \( t \) and \( T \) be the GEQ terms corresponding to \( t \) and \( T \) by mapping bcic \( \lambda \) to GEQ \( \lambda \), etc. Then \( \cdot \vdash t \iff T \vdash t : T \).

See proof in Appendix A.

### 5.4.3.3 Gradual Guarantees

To prove the gradual guarantees, we use the gradual criteria to show that elaboration is monotone. This, when combined with the monotonicity of \( \sim \) with respect to semantic precision, gives us both the static and dynamic guarantees.

**Lemma 5.4.20 (Elaboration Gradual Guarantee).** Suppose \( t_1 \subseteq_{\text{Surf}} t_2 \) and \( \Gamma_1 \subseteq \Gamma_2 \) (i.e. entries in \( \Gamma_1 \) and \( \Gamma_2 \) are respectively related by \( \subseteq \)). Then:

- If \( \Gamma_1 \vdash t_1 \Rightarrow t_2 \Rightarrow t_2 \Rightarrow T \text{ for some } t_2 \text{ where we have } \Gamma_1 \vdash t_1 \Rightarrow t_2 \).

- If \( \Gamma_1 \vdash t_1 \Rightarrow T_1 \) then \( \Gamma_2 \vdash t_2 \Rightarrow T_2 \text{ for some } T_2, t_2 \text{ where we have } \Gamma_2 \vdash T_2 \Rightarrow T_1 \Rightarrow Type \text{, and } \Gamma_1 \vdash t_1 \Rightarrow t_2 \).

See proof in Appendix A.

When combined with the preservation of precision under evaluation, this is enough to prove the static and dynamic gradual guarantees as stated in Section 5.4.1. The hard work lies in proving that reduction preserves precision, which we leave to Section 5.5.3.

### 5.5 Consistency and Precision

Motivated by the criteria of Section 5.4.2, in this section we extend GCIC’s precision and consistency relations to accommodate propositional equality and composition. We show that our relations fulfill the laws of Section 5.4.2, thus showing that GEQ fulfills type safety and the gradual guarantees, justifying the design of precision and consistency.
5.5.1 Precision and Consistency in gc1c

Figures 5.8 and 5.9 present structural precision from gc1c, written as $\Gamma \vdash t_1 \sqsubseteq t_2$. Structural precision is the syntactic relation out of which definitional precision $\sqsubseteq \_\_\_$ is built. The generating rules GENUNK and GENERR establish $\sqsubseteq_T$ and $\sqsubseteq_T$ as the least and most precise terms of type $T$. We include GENUNKUNIV for consistency with gc1c, allowing some cumulativity for $\sqsubseteq_{\text{Type}}$. The rule GENERRLAM encodes a version of $\eta$-expansion for errors. The diagonal rules (named $\text{Diag}^*$) are structural: terms are precision related if they are built with the same syntactic construct and the corresponding sub-terms are precision-related. Finally, cast rules capture non-structural properties of casts. Rule CAST-L states that a casting $t$ is more precise than $t'$ if the cast’s source and destination types are both more precise than the type of $t'$, and if $t$ is more precise than $t'$. The rule CAST-R says the opposite: casting $t$ is less precise than $t$ if the source and destination are both less precise than the type of $t$ and $t$ itself is less precise than $t'$.

To use structural precision in GEQ, we make one major change from the original presentation in gc1c. In place of type synthesis, we use presynthesis, written $\Gamma \vdash t \rightsquigarrow T$, which is defined to be exactly the type synthesis relation without the $\sqsubseteq \_\_\_$ side-condition in CASTREFL. Presynthesis types strictly more terms than synthesis, and both produce the same type, since they differ only in side-conditions. The side-condition is not used in the type-safety proof, so any runtime terms that presynthesize a type are safe. Unlike gc1c, GEQ uses precision to type equality witnesses, so presynthesis avoids a circular dependency between typing and precision.

Structural precision is defined mutually with definitional precision (Fig. 5.10) $\sqsubseteq \_\_\_$, which acts as $\sqsubseteq \_\_\_$ from Section 5.4. Definitional precision allows reducing before comparing, and is used with type ascriptions, such as for functions, equality proofs and casts. Since the checking rule for CASTCIC allowed arbitrary reductions, a term may be well typed even if its type ascriptions are not fully reduced. Type ascriptions on a term may need to be reduced before structural precision is apparent. This definition is due to Lennon-Bertrand et al. [87].
Γ ⊢ t₁ ⊑ₜ t₂ (Precision: Generating and Cast Rules)

<table>
<thead>
<tr>
<th>GenUnk</th>
<th>GenErr</th>
<th>GenUnkUniv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Γ₁ ⊢ t ⇒ * T'</td>
<td>Γ₁ ⊢ T ⇒ * T'</td>
<td>Γ₁ ⊢ T ⇒ * Type_i</td>
</tr>
<tr>
<td>Γ₁</td>
<td>Γ₂ ⊢ T' ⊑ₜ T</td>
<td>Γ₁</td>
</tr>
<tr>
<td>Γ₁</td>
<td>Γ₂ ⊢ t ⊑ₜ t'</td>
<td>Γ₁</td>
</tr>
</tbody>
</table>

CastL  
Γ₂ ⊢ t₂ ⇒ * T₂  
Γ₁|Γ₂ ⊢ T₁ ⊑ₜ T₂  
Γ₁|Γ₂ ⊢ t₁ ⊑ₜ t₂  
Γ₁|Γ₂ ⊢ (T₁' ⊑ₜ T₁) t₁ ⊑ₜ t₂

CastR  
Γ₁ ⊢ t₁ ⇒ * T₁  
Γ₁|Γ₂ ⊢ T₁ ⊑ₜ T₂  
Γ₁|Γ₂ ⊢ t₁ ⊑ₜ t₂  
Γ₁|Γ₂ ⊢ U₁ ⊑ₜ T₂  
Γ₁|Γ₂ ⊢ t₁ ⊑ₜ t₂  

GenErrLam  
Γ₂ ⊢ t₂ ⇒ * (x : T₂) → T₂'  
Γ₁|Γ₂ ⊢ (x : T₁) → T₁' ⊑ₜ (x : T₂) → T₂'  
Γ₁|Γ₂ ⊢ λ(x : T₁). U₂ ⊑ₜ t₂

Figure 5.8: Structural Precision: Generating and Cast Rules

5.5.2 Precision and Consistency for GEq

The structural precision laws are not sufficient for handling composition. In particular, we want Γ|Γ' ⊢ t₁ &ₜ t₂ ⊑ₜ t₁, with the same holding for t₂. However, this fact is not derivable from the diagonal rule for composition. Instead, we must add rules to ensure that composing produces a lower bound. However, once we start adding non-structural rules, we must be careful not to disrupt the other properties we need from precision. For example, Section 5.4.2 states that precision must be transitive. If (x &ₜ y) &ₜ z ⊑ₜ x &ₜ y and x &ₜ y ⊑ₜ x, but we also want (x &ₜ y) &ₜ z ⊑ₜ x, then we must transitively apply the fact that composing produces a lower bound.

Figure 5.11 shows the precision rules for GEq’s new language forms. DiagRefl, DiagComp, DiagEq and DiagJ are like the other diagonal rules. The rules PrecCompL and PrecCompR encode that the composition is a precision-lower bound, but in a way that preserves transitivity. The rules for ⊑ₜ, used to define valid witnesses, are given in Fig. 5.12: structural rules can be applied
\[ \Gamma \vdash t_1 \sqsubseteq_\alpha t_2 \quad (\text{Precision: Diagonal Rules for gcic forms}) \]

<table>
<thead>
<tr>
<th>Rule</th>
<th>Context</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DiagAbs</strong></td>
<td>( \Gamma_1 \vdash \Gamma_2 \vdash x \sqsubseteq_\alpha x )</td>
<td>( \Gamma_1 \vdash \Gamma_2 \vdash \lambda(x : T_1).t_1 \sqsubseteq_\alpha \lambda(x : T_2).t_2 )</td>
</tr>
<tr>
<td><strong>DiagCast</strong></td>
<td>( \Gamma_1 \vdash \Gamma_2 \vdash T_1 \sqsubseteq_\alpha T_2 ) \quad ( \Gamma_1 \vdash \Gamma_2 \vdash T'<em>1 \sqsubseteq</em>\alpha T'<em>2 ) \quad ( \Gamma_1 \vdash \Gamma_2 \vdash t_1 \sqsubseteq</em>\alpha t_2 )</td>
<td>( \Gamma_1 \vdash \Gamma_2 \vdash t_1 \sqsubseteq_\alpha t_2 ) \quad ( \Gamma_1 \vdash \Gamma_2 \vdash t'<em>1 \sqsubseteq</em>\alpha t'_2 )</td>
</tr>
<tr>
<td><strong>DiagInd</strong></td>
<td>( \Gamma_1 \vdash \Gamma_2 \vdash t_1 \sqsubseteq_\alpha t_2 )</td>
<td>( \Gamma_1 \vdash \Gamma_2 \vdash C^{@(i)}t_1 \sqsubseteq_\alpha C^{@(i)}t_2 )</td>
</tr>
<tr>
<td><strong>DiagApp</strong></td>
<td>( \Gamma_1 \vdash \Gamma_2 \vdash t_1 \sqsubseteq_\alpha t_2 ) \quad ( \Gamma_1 \vdash \Gamma_2 \vdash t'<em>1 \sqsubseteq</em>\alpha t'_2 )</td>
<td>( \Gamma_1 \vdash \Gamma_2 \vdash t_1 \sqsubseteq_\alpha t_2 ) \quad ( \Gamma_1 \vdash \Gamma_2 \vdash D^C(t_1,t'<em>1) \sqsubseteq</em>\alpha D^C(t_2,t'_2) )</td>
</tr>
<tr>
<td><strong>DiagUniv</strong></td>
<td>( \Gamma_1 \vdash \Gamma_2 \vdash t_1 \sqsubseteq_\alpha t_2 )</td>
<td>( \Gamma_1 \vdash \Gamma_2 \vdash \text{Type}<em>{\ell} \sqsubseteq</em>\alpha \text{Type}_{\ell} )</td>
</tr>
<tr>
<td><strong>DiagProd</strong></td>
<td>( \Gamma_1 \vdash \Gamma_2 \vdash T_1 \sqsubseteq_\alpha T_2 ) \quad ( \Gamma_1,(x : T_1) \vdash \Gamma_2,(x : T_2) \vdash T'<em>1 \sqsubseteq</em>\alpha T'_2 )</td>
<td>( \Gamma_1 \vdash \Gamma_2 \vdash (x : T_1) \rightarrow T'<em>1 \sqsubseteq</em>\alpha (x : T_2) \rightarrow T'_2 )</td>
</tr>
<tr>
<td><strong>DiagFix</strong></td>
<td>( \Gamma_1 \vdash \Gamma_2 \vdash s \sqsubseteq_\alpha s' )</td>
<td>( \Gamma_1 \vdash \Gamma_2 \vdash s \Rightarrow _ C(a) \quad \Gamma_1 \vdash \Gamma_2 \vdash s' \Rightarrow _ C(a') \quad \Gamma_1,(z : C(a)) \vdash \Gamma_2,(z : C(a')) \vdash P \sqsubseteq_\alpha P' \quad \Gamma_1,(f : (z : C(a))) \vdash \Gamma_2,(f : (z : C(a'))) \vdash t_k \sqsubseteq_\alpha t'_k )</td>
</tr>
</tbody>
</table>

**Figure 5.9:** Precision: Diagonal Rules for gcic forms
Figure 5.10: Definitional Precision

(PrecModStruct), and as with $\subseteq$, they allow for reductions before comparing with structural precision (PrecModStepL, PrecModStepR). They also allow backwards steps (PrecModStepBack), fulfilling Lemma 5.4.9. We only allow backwards-steps for the less precise term, since backwards steps on the left-hand are admissible by Lemma 5.4.13.

Last are the rules for (static) consistency for equality and composition (Fig. 5.13). Recall that Section 5.4.2 requires that reducing precision preserves consistency. Since composition is as precise as both its arguments, $t_1 \& t_2 \equiv t_3$ should imply that $t_1$ and $t_2$ are both consistent with $t_3$. We conjecture that composition is a (semantic) greatest lower bound, which would mean that errors in composing witnesses are never flagged earlier than necessary. For this to hold, the composition of two terms must be consistent with everything that is consistent with both of those two terms. Our composition consistency rules express this: CstCompL and CstCompR ensure that $t_1 \& t_2$ is consistent with exactly the terms that are consistent with both $t_1$ and $t_2$.

With consistency fully defined, the difference between static and dynamic consistency is now clear: two terms that share a non-error lower-bound may be statically inconsistent if they differ only in neutral terms. Variables are only statically consistent with themselves (CstVar) or $\_\_\_$ (CstUnkL,R). However, for any two variables $x$ and $y$, $x \& y$ is a non-error term that is as precise as both, as given by PrecComp(L,R). This disconnect between precision and consistency is justified by the criteria of Section 5.4.2: we show below that reducing precision preserves consistency. The separation of static and dynamic consistency enables the gradual guarantees and conservatively embedding $\text{CIC}$ while maintaining static equivalences.
Figure 5.11: Precision for Propositional Equality and Composition

Figure 5.12: Precision Modulo Conversion

Figure 5.13: Consistency for Propositional Equality and Composition
5.5.3 Fulfilling The Criteria

GEq is now fully defined: we have defined the precision relations $\sqsubseteq$, and $\sqsubseteq$, and the consistency relation $\equiv$, to instantiate $\sqsubseteq$, $\sqsubseteq$, and $\equiv$. We now establish that these relations fulfill the criteria of Section 5.4.2. We give the intuition behind some of the cases that are new compared to GCIC.

- **Immediate Results:** Proving reflexivity of $\sqsubseteq$ (Lemma 5.4.4) is a straightforward induction. The rules PRECCompL and PRECCompR make the composition of terms as precise as either term, proving Lemma 5.4.7. The closure of $\sqsubseteq$ under convertibility is built into its definition, proving Lemma 5.4.9. DiagCast gives that casts are monotone, proving Lemma 5.4.11. The monotonicity of substitution (Lemma 5.4.12) is proved with a straightforward induction, relying on presynthesis preserving types under substitution. The remaining diagonal rules give that $\sqsubseteq$ has all structural rules, fulfilling Lemma 5.4.15.

- **Composition Safety** (Lemma 5.4.5) For progress, each composition of two canonical forms of the same type has a reduction, where either the heads match and the arguments are composed, or an error is produced. If one of the composed terms is not a canonical form, then either (1) one of the composed terms can reduce, (2) one term is a $\mathtt{?T}$ or $\mathtt{U_T}$ where $T$ is not a function or equality type, and we can reduce with REDCompUnk or REDCompErr, or (3) one of the composed terms is neutral, and hence the composition is neutral. For preservation, either the result is immediate, or casts are inserted to ensure that types are preserved.

- **Composition Confluence** (Lemma 5.4.6) REDCompUnk(L,R) ensures that composing with $\mathtt{?T}$ only reduces when $\mathtt{?T}$ cannot reduce, avoiding a “diamond” problem.

- **Precision Transitive** (Lemma 5.4.8): We prove this after monotonicity of reduction, which lets us prove that precision-related types have precision-related terms, which is necessary to fulfill premises on terms’ types, such as in CastL and CastR. The rest is straightforward induction.
• **Static Consistency** (Lemma 5.4.10) The GEQ rules not present in gcic are for equality, which are trivially handled, and consistency rules for composition, for which the result vacuously holds since composition is not present in the static language.

• **Monotonicity of Reduction** (Lemma 5.4.13): The key fact is that, since PrecCompl and PrecCompR only have composition on the left, all the inversions in the gcic proofs are still valid for GEQ. The interesting case is when precision is derived using PrecCompl (PrecCompR is symmetric), and the composition reduces. The result is trivial for RedCompUnk(L,R) and RedCompErr(L,R). For the remaining cases, two terms with the same head are being composed, and the result is either U or another term with the same head. When U is produced the result is trivial. When a term with the same head is produced, the PrecCompl can be used with the appropriate diagonal rule. In the case that casts are present in the result of composition, CastL is used. The other notable case is when J reduces, where the result is derived using DiagCast.

• **Consistency Monotone** (Lemma 5.4.14) We first show that consistency is monotone on the left, then prove that it is symmetric to obtain monotonicity for both arguments. The case when $\equiv_\alpha$ is derived with CstCompR or CstCastR must be handled specially, since they each take an operand that can be any term. The trick is to unwrap the chain of CstCompR and CstCastR uses, use the induction hypothesis on the contained derivation, then re-apply CstCompR and CstCastR in the same order to obtain the result. When precision is derived with PrecCompl or PrecCompR, then consistency must have either been derived with CstCompDiag, in which case the result follows from the induction hypothesis, or with CstCompl or (symmetrically) CstCompR. For CstCompl, the premise gives that both composed terms are consistent with the right-hand term, yielding our result. The remaining cases are straightforward.
5.6 Discussion

5.6.1 Extensions Enabled by Equality

In addition to catching the kinds of bugs discussed in Section 5.2, we show some benefits of having propositional equality in \( \text{GE} \). Three new language features can be encoded using propositional equality, without augmenting the cast calculus: empty types, Axiom K, and indexed inductive type families. For type families, we discuss some limitations of the approach and workarounds for these limitations, showing how our cast calculus is expressive enough to pave the way for future improvements.

5.6.1.1 The Empty Type

Just as the gradual \( \mathbb{J} \) needed computation, eliminating the empty type has computational content in a gradual language. In static languages, the empty type \( \text{Empty} \) has no closed values, so either \( \text{Empty} \) contains no terms, or (for logically inconsistent languages) any such terms are non-terminating. The elimination function \( \text{exfalso} : (X : \text{Type}) \rightarrow \text{Empty} \rightarrow X \) produces a result of any type, given a value of the empty type. In a gradual language, however, \( ? \) and \( \emptyset \) can be used at any type, including the empty type. So a gradual \( \text{exfalso} \) must produce a value of type \( X \).

We again follow the goal of dynamically tracking constraints expressed by types. For the empty type, a value of type \( f : T \rightarrow \text{Empty} \) encodes the constraint that \( T \) should be impossible, and a branch built using \( \text{exfalso} \) should be unreachable. If \( f \) is applied to \( t : \text{Empty} \), created using \( ? \) or casts, then the constraint has been violated, and an error should be raised.

We can encode this behaviour by defining \( \text{Empty} \) to be:

\[
\text{true} =_{\text{E}} \text{false}
\]

and \( \text{exfalso} \) to be:

\[
\lambda X. \lambda t. \mathbb{J} (b. \text{if } b \equiv X) \text{ true } t
\]

The key is \( ?_{\text{Empty}} \) and \( \emptyset_{\text{Empty}} \) both evaluate to \( \text{refl}(_{\text{E}})-\text{true}=\text{false} \). So the only closed value of type \( \text{Empty} \) is a dynamic type error. Likewise, the eliminator \( \text{exfalso} \) casts
t to type $\mathcal{U}_{\text{Type}}$, before casting it to type $X$, so the result is always $\mathcal{U}_X$. Without adding any features to CASTEQ, the bug-finding described in Section 5.2 handles constraints encoded as logical negation.

5.6.1.2 Axiom K

Because $?_{t_1 \cong t_2}$ steps to $\text{refl}(t_1 \& T_{t_1 \equiv t_2})$, GEQ is in the class of dependently typed languages where $\text{refl}$ is the only constructor for equality. Composition can be used to derive a (gradual) proof of this uniqueness, even though no such proof can be derived in most static type theories [75]:

$$
K : (x : T) \to (pf : x =_T x) \to pf =_{x = x} \text{refl}(x)_{x \equiv x} \\
K x pf = \text{refl}(pf \&_{x = x} \text{refl}(x)_{x \equiv x})_{pf = \text{refl}(x)_{x \equiv x}}
$$

Axiom K can be used to prove that all equality proofs of a given type are equal [141], so our proof-irrelevant $J$ principle does not lose any expressivity, since any types parameterized by an equality proof can be rewritten with $K$. Also, Axiom K allows for conventional dependent pattern matching to be elaborated into inductive eliminators [69], providing a lightweight alternative to the cumbersome $\text{ind}$ form. The combination of Axiom K and function extensionality suggests a connection to Observational Type Theory (OTT) [6, 124] that warrants future exploration. Sadly Axiom K means that GEQ is incompatible with univalence and homotopy type theory [149], but we leave univalence-compatible gradual equality to future work.

5.6.1.3 Inductive Types

McBride [95] describes how, using propositional equality, indexed inductive families can be encoded. The main idea is, instead of having each constructor return different indices, each index is a parameter, and each constructor takes an equality proof that the parameter has the desired value. In the elimination principle, $J$ is used to rewrite the type of the returned value using the stored equality. Consider
how the classic vector type:

```
data Vec (X : Type) : (n : ℕ) → Type  where
   Nil : Vec X 0
   Cons : X → Vec X n → Vec X (1+n)
```

is transformed into:

```
data Vec' (X : Type) (n : ℕ) : Type  where
   Nil' : (n =₀ 0) → Vec X y
   Cons' : (z : ℕ) → X → Vec X z
             → n =₀ (1+z) → Vec' X n
```

This transformation gives a low-overhead way to incorporate indexed inductive families with gradual dependent types. Since no extensions to CAST Eq are required, the safety and gradual guarantee results from Section 5.4 apply. The constructors take equality proofs, so violations of those equalities raise dynamic type errors.

However, a naive transformation of inductive types is limited in its ability to eagerly detect errors. The problem is that dynamic consistency is fundamentally not transitive, since otherwise all types are consistent through ?. Members of inductive types are essentially trees, and equality constraints track constraints at each level of the tree, but consistency across the entire tree is not ensured. The witnesses track the evolution of type information across time, but not across space. Consider:

```
Cons’ ?₀ₜr true (Nil’ ?₀ₙ refl(0)·?₀ₙ ₀ₙ) refl(2)·?₀ₙ ₂ₙ : Vec’ ⊕ 2
```

a vector with one element, whose type says it has length 2. Constructing this vector raises no run-time type errors. At each level, the equality proof is correct: 0 is consistent with ?₀ₙ, and 2 is consistent with 1 + ?₀ₙ. Gradually, the non-transitivity means that imprecision at each level can cause disconnects between levels.

Thankfully, CAST Eq is expressive enough to encode a solution to this problem.
By having composition as an operator in the language, one can define so-called “smart constructors” that have the same types as the normal constructors, but that access the equality proofs stored in the previous level of the tree when constructing new ones. For example, using \( J \) we can write

\[
\text{cong1} : (m \, n : \mathbb{N}) \rightarrow m =_\mathbb{N} n \rightarrow 1 + m = 1 + n
\]

which can be used in a "smart" Cons:

\[
\text{smartCons} \; z \; h \; t \; \text{eq} = \\
\text{Cons} \; z \; h \; t \; (\text{eq} \; \&\&(1+z) =_\mathbb{N} n \leftrightarrow 1+z =_\mathbb{N} 1+z)(\text{cong1} \; (\text{wit} \; t))
\]

where \( \text{wit} : \text{Vec} \; X \; z \rightarrow z =_\mathbb{N} z \)

\[
\text{wit} \; (\text{Nil} \; \text{eq}) = \langle z =_\mathbb{N} z \leftrightarrow 0 =_\mathbb{N} z \rangle \text{eq}
\]

\[
\text{wit} \; (\text{Cons} \; x \; h \; t \; \text{eq}) = \langle z =_\mathbb{N} z \leftrightarrow 1 + x =_\mathbb{N} z \rangle \text{eq}
\]

When \( \text{smartCons} \) is used in place of \( \text{Cons}' \), the witness \( \text{refl}(0).?_\mathbb{N} =_\mathbb{N} 0 \) is transformed to \( \text{refl}(1).?_\mathbb{N} =_\mathbb{N} 1 \), which produces an error when cast to \( 1+_\mathbb{N} =_\mathbb{N} 2 \), since \( 1?_\mathbb{N} 2 \sim \mathbb{N} \). Formalizing the general version of this approach is beyond the scope of this chapter, but it shows how having composition as an operator enables more detailed manipulation of run-time type information.

### 5.6.2 Future Work

#### 5.6.2.1 Termination and Approximate Normalization

As presented, GEQ has undecidable type checking. Precision and CASTEQ typing are not decidable. Thankfully, we do not need to decide precision typing to decide GEQ typing, since initial witnesses are always valid by reflexivity. Deciding GEQ typing is straightforward, except for normalization: some terms do not terminate and consistency compares modulo reduction. In GCIC, Lennon-Bertrand et al. [87] show that termination can be obtained by sacrificing the gradual guarantees, or by restricting universes so that they are not closed under function types. While useful for type theory, these sacrifices remove reasoning principles or reduce expressivity, respectively, that make them unsuitable for programming. In
Chapter 6 we describe how to regain decidable type checking for GEq, adapting the techniques of approximate normalization from Chapter 3.

5.6.2.2 Conjectures: EP-Pairs, Composition, Blame and Full Abstraction

Lennon-Bertrand et al. [87] prove a stronger property than the gradual guarantees for gcic. They show that casts between precision-related types form an embedding-projection pair (EP-pair) [105], so that increasing then decreasing precision produces the same result modulo errors, and decreasing then increasing precision produces an observationally-equivalent result. While the gradual guarantees are helpful, they are satisfied by trivial languages where every cast produces ?. Showing the EP-pair property would prove that casts in GEq never lose run-time information, giving more confidence in its ability to dynamically track constraints. We conjecture that GEq fulfills the EP-pair property, but suspect novel proof techniques are needed to handle witness proofs. We also conjecture that composition computes the greatest lower-bound for semantic precision, so that each type forms a true semi-lattice. This would establish that witness composition never prematurely raises dynamic errors, since two witnesses would compose to \( \mathcal{U} \) only when all other options are impossible.

Another desirable property of gradual languages is a blame theorem [153], stating that, for a dynamic cast error, the less-precise type is always blamed. GEq has no notion of blame, but we conjecture that the techniques of Zalewski et al. [158] could be adapted to GEq.

Finally, we conjecture that there is a variant of cic whose embedding into GEq is fully abstract, meeting the criteria Jacobs et al. [81] set out for gradual languages. Intuitively, we can form equalities between extensionally-equal functions, and use those to cast between types indexed by them, so any property of a function should apply to one that is extensionally-equal. Full abstraction guarantees that all static equivalences hold in GEq, giving the programmer more tools with which to reason about their code. Proving full abstraction for non-dependently typed gradual languages is a recent development, so more investigation is needed to adapt these techniques to dependent types. The usual technique for full abstraction is to simulate the target language in the source, so every target
context can be translated into a source context that is unable to distinguish the terms. As a consequence, the embedded variant of CIC must have capabilities for non-termination added.

5.6.3 Related Work

5.6.3.1 Gradual Approaches to Equality

GRIP [90] extends CastCIC with propositional equality, but unlike GEq, this propositional equality is in a separate sort, the types of which have all members definitionally equal. This propositional layer has no imprecision, and contains features for catching dynamic type errors in the gradual layer. GRIP features an internal notion of precision: while not all terms obey the gradual guarantees, self-precise terms do, and guarantees for such terms are available as a theorem in the propositional layer. GEq provides reasoning about equality within gradual programs, whereas GRIP provides a layer for reasoning about gradual programs. GRIP is based on OTT, and hence supports fully static proofs of function extensionality and Axiom K. GEq contains only gradual proofs of these. We conjecture that, like in GRIP, OTT could replace BCI C as the underlying static language for GEq.

Our work also relates to that of Lemay [85], which presents a dependently typed language where all definitional equality checks are deferred to run time. Like in gccic and GEq, casts are inserted when a term’s synthesized type differs from the type against which it is checked. However, there is no unknown term/type, and no requirement that synthesized and checked-against types be consistent, so errors are raised only when safety is directly violated. Like in GEq, function equality is checked extensionally: casts accumulate on unapplied functions, and arguments are cast when cast-containing functions are applied, similar to composing functions in GEq. The work emphasizes clear error messages, and unlike GEq, features a rich notion of blame.

5.6.3.2 Flexible Dependent Types

GEq builds on a long line of work mixing dynamic and static enforcement of specifications. Ou et al. [114] support mixed static and dynamic checking of boolean-
valued properties, and Lehmann and Tanter [84] provide gradual typing for refinement types. Similarly, Tanter and Tabareau [144] develop a system of casts for Coq, using an unsound axiom to represent type errors. These casts worked for values of subset types, i.e. a value paired with a proof that some boolean-valued function returns true for that value, but not general inductive types. Osera et al. [113] present dependent interoperability for principled mixing of dependently typed and non-dependently typed programs. Dependent interoperability was extended by Dagand et al. [42, 43], who provide a general mechanism for lifting higher-order programs to the dependently typed setting. All of these approaches presuppose separate simple and dependent versions of types, related by boolean-valued predicates. Our composition of witnesses provides similar checks, but by keeping witnesses, types need not be reformulated in terms of subset types or boolean predicates.
Chapter 6

A Syntactic Model of Approximate and Exact Gradual Semantics

In the previous chapters, we saw a gradual language with decidable type checking but without inductive types, a cast-calculus based language with inductive types but without decidable type checking, and a method for adding propositional equality into that cast calculus. Now, we combine the three of these into a language with decidable type checking, that supports inductive types, and propositional equality, without sacrificing the gradual guarantees or the expressivity of cic.

The resulting language, \( \text{GEQ} \), is nearly identical to GEQ, with the only significant difference being how casts from function types to ? are handled during type checking. However, the difficulty lies in showing that type checking is decidable, i.e. that all compile-time computations terminate. We do this with a syntactic model: we write a translation from \( \text{GEQ} \) to a static type theory. If each step of computation during type checking in \( \text{GEQ} \) corresponds to at least one step of reduction in a strongly-normalizing language, then all terms must terminate.

The syntactic model approach is useful for showing termination of approximate normalization, but it can also be used to model exact execution, using a static target with a modality for unbounded recursion. Modelling the exact semantics
lays the groundwork for proving additional metatheory about $\triangleright GEQ$ in the future. The model for exact execution is nearly identical to the model for approximate normalization. Moreover, it suggests an implementation strategy, since terms can be normalized or compiled by first translating to a static type theory.

### 6.1 Introduction

So far, the story of gradual dependent types has been one of tension and compromise. Chapter 3 explored the tension between types depending on values and the effects introduced by gradual code, with approximate normalization being the compromise necessary to obtain both. Chapter 5 handled the inherent tension between preservation of static equivalences and conservative extension of $\text{CIC}$, balancing the two by separating static and dynamic notions of equality.

Lennon-Bertrand et al. [87] present a deeper conflict: the “fire triangle” of gradual dependent types. They show that no gradual language can conservatively extend $\text{CIC}$, have all terms strongly normalizing (which implies decidability), and fulfill the EP-pairs property (which implies the gradual guarantees). They present three calculi, each of which fulfills two of the above properties.

This chapter identifies a potential compromise to resolve the tension between conservatively extending $\text{CIC}$, decidable type checking, and the gradual guarantees. We present a Guarded Gradual language with propositional Equality and decidable type checking ($\triangleright GEQ$), which is nearly identical to $GEQ$, except that it features approximate normalization to ensure decidable type checking. We do so without restricting terms in the ways that prevent $GCIC$ variants from fulfilling the gradual guarantees or conservatively extending $\text{CIC}$, and we conjecture that $\triangleright GEQ$ fulfills both these properties.

Just as in Chapter 3, $\triangleright GEQ$ has separate approximate and exact semantics for compile time and run time respectively, which enable decidable type checking without losing expressivity. The difficult part is actually proving that approximate normalization terminates. The hereditary substitution strategy of Chapter 3 does not scale to inductive types, since eliminating a value from an inductive type is not structurally decreasing in the type of the variable being replaced.

Instead, we prove normalization using a syntactic model in the style of Bernardy
et al. [11], Boulier et al. [17]. That is, we show that every term has a normal form by translating said term to a term in Martin-Löf Type Theory (MLTT), which is strongly normalizing. By showing that each source reduction step corresponds to at least one reduction in the target, we prove that the number of source reductions must be finite.

The choice to use a syntactic model influences the design of approximate normalization. We use the cast-calculus approach of GCIC and GEQ rather than the evidence approach of GDTL because it integrates well with a syntactic model: source casts can be implemented as target functions. Instead of approximating based on the GDTL metric of arrow types at a given universe level, we approximate casts from function types to ?. This allows ? to be modelled as a strictly-positive inductive type, and hence allows it to be modelled in a strongly normalizing calculus.

6.1.1 Three Problems, One Solution

Along with our syntactic model for approximate normalization, we present a model of exact execution. We do this because the model is nearly identical: by parameterizing over whether we are in approximate or exact mode, we are able to reuse the vast majority of the machinery from our approximate model for the exact model. We do not use the exact model to prove any theorems other than soundness of the model with respect to the reduction semantics of GEQ. Nevertheless, we provide it in anticipation of two future benefits: mechanized metatheory and implementation.

6.1.1.1 Proving metatheory and reasoning about gradual programs.

Lennon-Bertrand et al. [87] provide a syntactic model for GCIC, but they do so by extending cic with non-positive datatypes. The resulting target type theory is logically inconsistent, and hence, theorems proven in it cannot necessarily be trusted. Their proof of the gradual guarantee relies on a complex simulation-based argument, which GEQ inherits in Chapter 5.

By translating GEQ to Guarded Type Theory (GTT), we are able to describe non-termination for its exact semantics, while still having a target which is logically consistent. So one can prove metatheory about GEQ in GTT. The denota-
tional nature of the model means that convertible terms are propositionally equal, so we can prove properties about $\triangleright GEQ$ terms by reasoning about $\triangleright GEQ$ values. This ability enables future developments to avoid the complicated simulation arguments used with $GCIc$ and $GEQ$.

6.1.1.2 Implementing conversion checking and compilation.

Translating $\triangleright GEQ$ to $GTT$ provides a path towards implementation. Eventually, we would like a fully implemented compiler for a gradual dependently typed language, but writing a compiler for a dependently typed language is no easy task, even without gradual types. However, through the Curry-Howard correspondence, $GTT$ is a usable programming language, and is very similar to the core languages of proof assistants like Agda and Idris. These languages already feature efficient procedures for comparing terms for convertibility, and for translating terms to machine code. So instead of writing a new compiler for gradual dependent types, we can use the translation given by our syntactic model to translate to the core languages of these proof assistants. Then, gradual terms can be compared for convertibility by comparing their translations, and can be compiled to machine code through the translation.

Moreover, using approximate normalization and translating to $GTT$ lets us compile to static dependently typed languages without altering their core type theories. Languages like Idris and Agda allow for terms to be marked as total, which makes the type checker skip termination checking for said definition. This feature is useful for definitions that are proven to be terminating with proofs that cannot be expressed in type theory. However, gradual typing is morally different: since some terms really do not terminate, marking them as terminating would be a lie. Instead, Agda and Idris\(^1\) have a weaker feature, where a definition can be marked as partial. Partial definitions are conservative: they are not normalized during type checking, so some safe programs may be rejected if partial values are convertible but not syntactically identical. $GTT$ as a theoretical model for this way

\(^1\)In Idris 1 had this behaviour, but Idris 2 has yet to implement this behaviour, and happily runs the type checker forever when comparing non-terminating terms. Safety can be recovered by defining partial functions in their own module, and exposing them with `export` rather than `public export`.
of representing non-termination, and producing well typed \( \text{gtt} \) terms shows that we could also produce well typed Agda or Idris terms by marking exact definitions as partial.

### 6.2 A Birds Eye View of the Syntactic Model

To begin, we present the various parts of the syntactic model of \( \text{\textgreater GEQ} \), giving the intuition for what purpose they serve and how they fit together.

#### 6.2.1 The Source Language

We do not describe the entirety of \( \text{\textgreater GEQ} \) here, since it very similar to GEQ (or specifically, CastEQ), as described in Chapters 4 and 5. We present \( \text{\textgreater GEQ} \) as a cast calculus, though it could be the target of elaboration from a surface language similar to GEQ. The main difference between \( \text{\textgreater GEQ} \) and CastEQ is that \( \text{\textgreater GEQ} \) has two separate reduction relations, \( \sim_{\text{Approx}} \) and \( \sim_{\text{Exact}} \), where the typing rules only ever use \( \sim_{\text{Approx}} \). The relation \( \sim_{\text{Exact}} \) is identical to \( \sim \) in GEQ, except that reductions in equality witnesses use \( \sim_{\text{Approx}} \). The approximate reduction rules \( \sim_{\text{Approx}} \) are the same as the corresponding GEQ reductions, except for those we highlight in Section 6.2.1.3. Likewise, there are two pairs of typing relations, \( \Gamma \vdash t \Rightarrow_{\text{Approx}} T \) and \( \Gamma \vdash t \Leftarrow_{\text{Approx}} T \) versus \( \Gamma \vdash t \Rightarrow_{\text{Exact}} T \) and \( \Gamma \vdash t \Leftarrow_{\text{Exact}} T \). However, even \( \Gamma \vdash t \Leftarrow_{\text{Exact}} T \) uses \( \sim_{\text{Approx}} \) to define convertibility.

In general, we use the metavariable \( \alpha \) to range over \{Approx, Exact\}.

#### 6.2.1.1 Typing Modes

Other than the changes we mark below, the typing rules of \( \text{\textgreater GEQ} \) follow GEQ. However, each typing judgment has an approximate and exact version. In general, each rule has two identical copies, where the approximate/exact mode of the conclusion matches the mode of each parameter. The main exception is that typing rules for type formers only have approximate forms, and only ever use approximate forms in their premises. Also, equality witnesses are always typed with the approximate rules. For example, consider the rules for conversion, casts
and equality proofs:

\[ \Gamma \vdash t \Rightarrow_{\text{approx}} T' \]
\[ T \xrightarrow{\text{approx}} T'' \quad T' \xrightarrow{\text{approx}} T'' \]
\[ \Gamma \vdash t \Leftarrow_{\text{approx}} T \]

\[ \Gamma \vdash T_j \Rightarrow_{\text{Type approx}} \text{Type}_1 \text{ for } j \in \{1, 2\} \]
\[ \Gamma \vdash t \Leftarrow_{\text{approx}} T_j \]
\[ \Gamma \vdash \langle T_2 \Leftarrow T_1 \rangle t \Rightarrow_{\text{approx}} T_2 \]

\[ \text{CastCheck} \]
\[ \text{CastCast} \]

\[ \text{CastRefl} \]
\[ \Gamma \vdash t_w \Rightarrow_{\text{approx}} T \quad \Gamma \vdash t_1 \Leftarrow_{\text{approx}} T \quad \Gamma \vdash t_2 \Leftarrow_{\text{approx}} T \]
\[ \Gamma \mid \Gamma \vdash t_w \subseteq t_1 \quad \Gamma \mid \Gamma \vdash t_w \subseteq t_2 \]
\[ \Gamma \vdash \text{refl}(t_w), t_1 \Rightarrow_{\text{approx}} t_1 =_{\text{approx}} t_2 \]

In CastCheck, approximate reductions are used, regardless of the mode. Likewise in CastCast, the source and destination types are checked in approximate mode regardless of what mode the whole term is synthesizing in. For CastRefl, endpoints and witnesses are checked in approximate mode, even for a term in exact mode. One final simplifying assumption we make is that CastFun has both types at the same level\(^2\).

6.2.1.2 Tagged Terms

In GCIC, a cast from germ\( h \) to ?\text{Type}\(_j\) does not reduce, but instead acts as a tag for a value in type ?. To avoid confusion between the computation of casts and the tagging of values in our model, we introduce two new explicit forms:

\[ t +:= \langle h_i \rangle t \mid \langle D_i^C \rangle (\overrightarrow{n}) \]  
\text{(Additional term forms for tagged terms)}

The first form is a value from the germ of \( h \) injected into ?\text{Type}\(_j\). For example, if \( p : 1 =_{\text{N}} 1 \), then \( (=_{0}) p : ?\text{Type}_0 \): the equality proof is injected into ? by tagging it with the head and level of its type. The second form is a data constructor \( D^C \) with a sequence of fields, injected into ?\text{Type}\(_j\). For each field the tag \( n \) stores the maximum number for which the first \( n \) function domain types of the field’s type do not contain ?\text{Type}\(_j\). This information is used to direct how such a term is approximately

\(^2\)Our Agda model has a notion of cumulativity, but we omit it to reduce the cognitive load of reading this chapter.
normalized: to embed a member of an inductive type into \(?\), each field has to be cast to type \(?\), or to a function producing \(?\) whose domain does not contain \(?\). For example, consider the inductive type which stores a proof that a type contains at least as many terms as the naturals:

\[
\text{data Infinite } (A : \text{Type}) : \text{Type} \text{ where } \\
tofrom : (f : A \rightarrow \mathbb{N}) \rightarrow (g : \mathbb{N} \rightarrow A) \rightarrow ((x : \mathbb{N}) \rightarrow f (g x) =_\mathbb{N} x) \rightarrow \text{Infinite } A
\]

Given \(\text{tofrom } f g p : \text{Infinite } A\), we can inject it into \(?\) as:

\[
\langle \text{tofrom}_0^{\text{Infinite}} \rangle (f, 0, (g, 1), (p, 1))
\]

Both \(g\) and \(p\) are functions with \(\mathbb{N}\) as their domain, so they can be losslessly embedded into \(?\) by casting to \(\mathbb{N} \rightarrow ?_{\text{Type}_\ell}\), without violating strict positivity. However, when injecting an \(\text{Infinite}\) proof into \(?\), the parameter \(A\) is set to \(?_{\text{Type}_\ell}\), since that is the least precise instantiation of this inductive type. So \(f\) has to be stored as part of an element of \(?\) by casting to \(?_{\text{Type}_\ell} \rightarrow \mathbb{N}\), which would violate strict positivity. So the 1 tags on \(g\) and \(p\) denote that the first type to the left of the arrow needs no approximation, while the 0 tag denotes that for \(f\), none of the types left of the arrow are guaranteed to be free of \(?\). To embed \(f\) as a field in a value of type \(?\), it must be converted to \(?\), which approximates it to the least precise value in its image.
The corresponding typing rules for the new forms are as follows:

\[ \text{CastTag} \]
\[ \Gamma \vdash t \xleftarrow{\approx} \text{germ}_{\ell}(h) \quad h \notin \{\Pi\} \cup \{C \mid C \in \text{TypeCTORS}\} \]
\[ \Gamma \vdash \langle h \rangle t \Rightarrow \approx \text{Type}_{\ell} \]

\[ \text{CastTagPiApprox} \]
\[ \Gamma \vdash t \xleftarrow{\approx \text{Type}_{\ell}} \quad \Gamma \vdash (\Pi \ell) \lambda x. t \Rightarrow \approx \text{Type}_{\ell} \]

\[ \text{CastTagPiExact} \]
\[ (x : \text{Type}_{\ell}) \Gamma \vdash t \xleftarrow{\text{Exact} \text{Type}_{\ell}} \]

\[ \text{CastTagInd} \]
\[ (x_1 : T_1) \rightarrow \ldots \rightarrow (x_n : T_n) \rightarrow T := \text{Args}_{\ell}(C, i, D)\{\text{Args}_{\ell}(C)\} \]
\[ \text{does not contain } \text{Type}_{\ell} \]
\[ n_k \text{ maximal} \]
\[ \Gamma \vdash t_k \xleftarrow{\approx} (x_1 : T_1) \rightarrow \ldots \rightarrow (x_n : T_n) \rightarrow \text{Type}_{\ell} \]
\[ \Gamma \vdash \langle D^C_s \rangle(t_k, n_k) \Rightarrow \approx \text{Type}_{\ell} \]

A term in \text{Type}_{\ell} consists of a head and a level, along with a term from the germ for that head. The separate rules \text{CastTagPiApprox} and \text{CastTagPiExact} handle the function case. The exact rule works as in GEQ, but the approximate rule has two extra constraints. First, the function must be \(\eta\)-expanded (because the rule only applies to \(\lambda\) terms), which we need for technical reasons. Second, the function must not refer to its argument.

For inductive types in \text{Type}_{\ell}, the rule \text{CastTagInd} says that an inductive member of \text{Type}_{\ell} is one where every non-function field has type \text{Type}_{\ell}. A field with a function type has its parameter types taken from the germ of \(C\), and its return type is \text{Type}_{\ell}. Any dependencies between fields are erased: if the germ of \(C\) has one field’s type depend on a previous field’s value, when converting to \text{Type}_{\ell} we use \(\approx\) for the previous field’s value. Also, only types that are “smaller” than \text{Type}_{\ell} can be used in the domains of function-typed fields. This rule is essentially to ensure that parameterized inductive types cannot be used to circumvent \text{CastTagPiApprox}, which in turn helps us prove strong normalization later on. So we specify that for each field, we use the highest value of \(n\) such that the domain types does not contain \text{Type}_{\ell}. Any functions whose domains refer to \text{Type}_{\ell} are
converted directly into \( \text{Type}_t \), i.e. \( n \) is set to 0.

In addition to the modified typing rules, we change the cast reduction rules to produce these tagged terms instead of using casts as tags:

\[
\text{RedTagFunGerm} \\
(\langle ?\text{Type}_t \leftarrow (x : T_1) \rightarrow T_2) t \xrightarrow{\text{Exact}} (\Pi)_1(\langle ?\text{Type}_t \rightarrow T_2) t)
\]

\[
\text{RedTagIndGerm} \\
(\langle x_1 : T_1 \rightarrow \cdots \rightarrow x_n : T_n \rightarrow T := \text{Args}_{k}(C,i,D)[t_{\text{par}},t_{\text{arg}}] \mid \langle x_1 : T'_1 \rightarrow \cdots \rightarrow x_n : T'_n \rightarrow T' := \text{Args}_{k}(C,i,D)[?\text{Params}_i(C),?\text{Args}_i(C)] \}
\]

\[
\langle ?\text{Type}_t \leftarrow C(t_{\text{par}})D^C(t'_{\text{par}},t_{k}) \rangle \xrightarrow{\text{\&}}^k
\]

\[
\langle D^C((\langle (x : T_1) \rightarrow \cdots \rightarrow (x : T'_n) \rightarrow ?\text{Type}_t \leftarrow (x : T_1) \rightarrow \cdots \rightarrow (x : T_n) \rightarrow T) t_k, n_k ) \}
\]

\[
\text{RedTagEqGerm} \\
(\langle ?\text{Type}_t \leftarrow t_1 \equiv T t_2) t \xrightarrow{\text{\&}}^k (\langle ?\text{Type}_t \equiv ?\text{Type}_t \equiv t_1 \equiv T t_2) t)
\]

\[
\text{RedTagUpDown} \\
\text{headMatch}(h,t)
\]

\[
(\langle T \leftarrow ?\text{Type}_t \rangle (h_t) t \xrightarrow{\text{\&}}^{\text{fromGerm}}(h,t)
\]

\[
\text{RedTagIndUpDown} \\
(\langle x_1 : T_1 \rightarrow \cdots \rightarrow x_n : T_n \rightarrow T := \text{Args}_{k}(C,i,D)[t_{\text{par}},t_{\text{arg}}] \mid \langle x_1 : T'_1 \rightarrow \cdots \rightarrow x_n : T'_n \rightarrow T' := \text{Args}_{k}(C,i,D)[?\text{Params}_i(C),?\text{Args}_i(C)] \} \)

\[
\langle C(t_{\text{par}}) \leftarrow ?\text{Type}_t \rangle \langle D^C((l_{k}, n_k)) \rangle \xrightarrow{\text{\&}}^k
\]

\[
D^C(t_{\text{par}},(\langle (x : T_1) \rightarrow \cdots \rightarrow (x : T_n) \rightarrow T \leftarrow (x : T'_1) \rightarrow \cdots \rightarrow (x : T'_n) \rightarrow ?\text{Type}_t \rangle t_k ) \}
\]

To cast to \( ? \), we cast to the germ, then tag that value. To cast from \( ? \), we get the tagged value, which is a member of the germ type, then cast that to the desired target type. We do so using an (omitted) judgment headMatch, which checks if \( t \) was constructed with a term former matching \( h \), and a metafunction fromGerm that behaves just like the GEQ cast from a germ type to a type. Essentially, RedTagUpDown behaves like RedCastUpDown in GEQ, but does not reduce if the term being cast is neutral, which helps our termination argument. For inductives,
we cast each field to or from ?i, or to/from a function type returning ?i. We save the reduction that approximately converts functions to ?i until Section 6.2.1.3, but all other reductions from GEQ work both approximately and exactly without modification.

### 6.2.1.3 Approximating Functions

The main difference in the definition of \( \leadsto_{\text{Approx}} \) is that casting from \((x : T_1) \to T_2\) to \(?_i\) approximates in \(?_i\) where it did not in GEQ. The reduction rule RED-CastFunGerm is replaced with:

\[
\text{RedCastFunGermApprox}
\begin{align*}
(x : T_1) \to T_2 &\neq \text{germ}_j(\Pi) \text{ for } j \geq i \\
(\text{Type}_i \leftarrow (x : T_1) \to T_2) &\leadsto_{\text{Approx}} \\
(\Pi_i)(?\text{Type}_i \to ?\text{Type}_i &\leftarrow (x : T_1) \to T_2) \lambda y.([y/x]T_2 \leftarrow [?T_1/x]T_2) (t \ ?T_1)
\end{align*}
\]

That is, a function is cast to \(?_i\) by casting through the germ, but instead of casting the function itself, we use the constant function producing the original function’s least precise value. Because the function type might be dependent, we must cast the result from the codomain for input \(?_i\) to the codomain for the actual input. Notice how this approximation is less lossy than in GDTl (Section 3.5.2.1): we approximate the input of the function to \(\?\) rather than the output. For example, \(\lambda(x : \text{Nat}). S \ x\) is approximated to \(\lambda(x : \text{Nat}). S \ ?\text{Nat}\), retaining the information that the result of the function is not zero. Constant functions are not approximated at all.

### 6.2.1.4 Terms in Types

We must approximate terms when they cross the boundary between terms and types. Types always use the approximate judgment, and the approximate and exact typing judgments are different. So, when substituting within a type we
must use an approximate version of a term:

\[
\text{CastApp} \quad \begin{align*}
\Gamma \vdash t & \Rightarrow \Pi (x : T_1) \rightarrow T_2 \\
\Gamma \vdash t & \Leftarrow T \\
\Gamma \vdash t_0 \vdash t_1 & \Rightarrow [\text{toApprox}(t_1)/x]T_2
\end{align*}
\]

Here the toApprox metafunction traverses a term and replaces all instances of \(\langle \Pi \ell \rangle t\) with \(\langle \Pi \ell \rangle (\lambda x . t \ ? \text{Type}_\ell)\).

### 6.2.2 What’s In a Model?

Now that we have appropriately modified GEQ into & GEQ, we can discuss how to model it in a static type theory. Our approximate and exact models of & GEQ can each be broken into the following parts:

- A type \(\mathbb{C} \ell\) of codes, representing the & GEQ Type\(_\ell\) for each \(\ell\);
- A translation \(\mathcal{T}[[\_]]\) mapping each & GEQ type to a code;
- An “elements-of” function \(\mathcal{E} : \mathbb{C} \ell \rightarrow \text{Type}_\ell\) which interprets codes into target types;
- A translation \(\mathcal{E}[[\_]]\) which maps each & GEQ term into a description of a computation in the target, under some monad \(\mathcal{L}\). For approximate normalization, this monad is the identity;
- A proof that for each \(t : T\), that \(\mathcal{E}[[t]] : \mathcal{L} (\mathcal{E}[[\text{Type}]]\))
- A proof that, for each reduction rule, applying \(\mathcal{E}[[\_]]\) to the redex and reduct produces propositionally equal terms
- A proof that, for each approximate reduction rule, applying \(\mathcal{E}[[\_]]\) to the redex produces a target redex

The last property is necessary to show termination of approximate normalization: if the number of & GEQ reductions is bounded by the number of target reductions, then we know the number of source reductions is finite, since the target is strongly
normalizing. This proof must be completed in the metalanguage, rather than in the target type theory. All convertible terms are definitionally equal, so there is no general way to express in a type theory that a term of that theory reduces to another term using one or more steps.

For the model, we need a target term for each source term. Each introduction form in the >GEq corresponds to an introduction form (usually a constructor or function abstraction) in the target. Each elimination form in >GEq corresponds to a function that we construct in the target, whose type matches the typing rule for the eliminator (albeit, producing a result under L). So the hard work of the model lies in constructing the types that denote values, and in constructing the supporting functions that correspond to elimination forms. These two tasks are interrelated: how we construct the type for values influences what operations we can write for those values. For the approximate case, the challenge of proving termination reduces to constructing well typed elimination functions in the target.

6.2.3 The Target Language

To establish our model, we translate >GEq to GTT. However, GTT comes in many flavours, providing similar core operations with slightly different implementations and metatheory. For our model, we use a variant of GTT called Ticked Cubical Type Theory (TCP) [101]. We restrict our model to the subset of TCP without glue types or systems, extended with inductive-recursive datatypes. We rely heavily on function extensionality, but we do not rely on axiom K, nor do we use univalence or homotopy type theory [149]. The details of TCP are not needed to understand our model, other than the fact that it gives a logically consistent language with the primitives we need for guarded recursion, and that it has a relatively faithful implementation in Guarded Cubical Agda [151].

In general, we only give a sketch of the actual TCP terms we construct in our model. The terms themselves are long, and a large portion of the development is devoted to proving inequalities about the sizes of terms that are immediately apparent to a human, or proving lemmas about transporting terms between various equal types. In the name of clarity, we use Agda-style inductive and record type definitions, and omit implicit arguments when they negatively impact the
readability of the given terms. We have mechanized our translation in Guarded Cubical Agda. The interested reader can find the complete translation archived on Zenodo [56].

6.2.3.1 The Exact Target

The primitives of \( \texttt{GTT} \) (implemented by \( \texttt{TCTT} \)) that we use are as follows:

\[
\begin{align*}
\triangleright &: \ Type_\ell \rightarrow Type_\ell \\
\triangleright &: \ Type_\ell \rightarrow Type_\ell \\
\text{mapNext} &: (f : A \rightarrow B) \rightarrow (x : A) \rightarrow (\text{next } f)(\text{next } x) = B \text{next } (f \ x) \\
\text{dfix} &: (\triangleright A \rightarrow A) \rightarrow \triangleright A \\
\text{pfix} &: (f : \triangleright A \rightarrow A) \rightarrow \text{dfix } f = \text{next } (f (\text{dfix } f)) \\
\text{Eq}_{\triangleright} &: (f : \triangleright Type_\ell \rightarrow Type_\ell) \rightarrow \triangleright (\text{dfix } f) = \triangleright (f (\text{dfix } f))
\end{align*}
\]

The guarded modality \( \triangleright \) describes a type whose values are obtainable “one step” into the future, i.e. after unfolding a guarded fixed point once. Along with this modality, an operation \( \triangleright \) turns a guarded type into a type. For any guarded type \( T : \triangleright Type_\ell \), the type \( \triangleright T \) contains values available one step in the future from a type available one step in the future. The guarded modality is applicative: next lifts any value into the guarded modality, and \( \triangleright \) applies guarded functions to guarded values. However, it is not monadic: a value of type \( \triangleright\triangleright T \) generally cannot be converted into one of type \( \triangleright T \), since a value obtainable after two steps is not necessarily obtainable after one step.

The power of \( \texttt{GTT} \) comes from its fixed-point operations: \( \text{dfix} \) finds the fixed point of any function, provided it only refers to its argument behind the guarded modality. \( \text{pfix} \) establishes that \( \text{dfix} \) actually computes a fixed point, and \( \text{Eq}_{\triangleright} \) lets us get rid of \( \triangleright \) in the fixed-points of types. The equality \( \text{Eq}_{\triangleright} \) protects \( \texttt{TCTT} \) from non-termination: it is an axiom, rather than a defined equality term, so transporting with it does not reduce. This allows possibly-nonterminating computations to
be propositionally equal to their later states without having any TCTT terms run forever.

From these primitives, we can build functions that are easier to use, and more readily show the usefulness of the guarded modality. For example, we can construct a fix operator as follows:

\[
\begin{align*}
\text{fix} : (f : A \rightarrow A) &\rightarrow A \\
\text{fix } f &\equiv f (\text{dfix } f)
\end{align*}
\]

The fix operator lets us take a “fixed point” of any guarded function and obtain a result that is not behind the guarded modality, that obeys a guarded version of the usual fixed-point equation. There are a few other helpful functions we can construct, whose types we give below:

\[
\begin{align*}
\text{l"ob} : (f : A \rightarrow A) &\rightarrow \text{fix } f =_A f (\text{next } (\text{fix } f)) \\
\text{tyFix} : (f : \text{Type} \rightarrow \text{Type}) &\rightarrow \text{Type} \\
\text{tylob} : (f : \text{Type} \rightarrow \text{Type}) &\rightarrow \text{tyFix } f =_{\text{Type}} f (\langle \text{tyFix } f \rangle)
\end{align*}
\]

The l"ob function is called L"ob-induction, and it lets us reason about an arbitrary number of unwrappings of fix. Likewise, tyFix says that we can turn a function between types into a type, with tylob establishing that any self-reference is behind the guarded modality. tyFix lets us define non-positive inductive datatypes so long as all non-positive references are behind the guarded modality, which is critical when exactly modelling ? as an inductive type.

### 6.2.3.2 The Approximate Target

The approximate version of our model uses the subset of TCTT that is equivalent to MLTT with inductive-recursive types and an axiom for function extensionality. MLTT with extensionality and induction-recursion is known to be strongly normalizing [53]. Because no guarded primitives are used in the approximate model, each \text{GEQ} term’s reductions are simulated by some MLTT term’s reductions. On the other hand, the exact translation produces a description of the term’s computation, but reductions are only reflected as propositional equalities.
6.2.4 Mode Tags

To avoid redundancy in our model, we often parameterize over whether we are in "approximate" or "exact" mode. To facilitate this, we define the following type of tags:

```haskell
data A where
    Approx : A
    Exact : A
```

We commonly use the variable name \( \alpha \) to describe values of type \( A \).

6.2.5 Modelling Values and Expressions

6.2.5.1 Codes and Interpretations

At the core of our model is a Universe à la Tarski, that is, a type that defines codes for \( \rhd \text{GE} \) language types, and an interpretation function that takes each code to a target type. Formally, we define a type \( C : \mathbb{N} \rightarrow \text{Type} \), and an "Elements-of" interpretation \( \text{El} : (\alpha : A) \rightarrow (\ell : \mathbb{N}) \rightarrow C \ell \rightarrow \text{Type} \). Here \( \ell \) corresponds to the \( \rhd \text{GE} \) universe-level of a given type, which we omit from \( \text{El} \) when it is clear from context. Likewise, we omit \( \alpha \) when it is clear from context, or when we are being generic over any tag. Using codes instead of \( \text{tctt} \) types is critical for gradual typing, because we need to access run-time type information that is not accessible from target types. Unlike target types, we can pattern match on codes and inspect what constructors were used to form a code. Also, all codes are able to reside in the \( \text{tctt} \) universe \( \text{Type}_0 \), regardless of the source \( \ell \) they model, which simplifies the development in some places.

6.2.5.2 A Lifting Monad

Codes can model \( \rhd \text{GE} \) types and their values, but since some \( \rhd \text{GE} \) terms are non-terminating, they cannot model all run-time computations. Even the guarded modality can only model computations one time-step into the future. To describe
possibly non-terminating computations, we use a \textit{guarded lifting monad} \cite{117}:

\[
\text{data } \mathcal{L} (A : \text{Type}) : \text{Type} \text{ where } \\
\text{Now} : A \rightarrow \mathcal{L} A \\
\text{Later} : \triangleright(\mathcal{L} A) \rightarrow \mathcal{L} A
\]

The lifting monad describes computations requiring some number of recursive foldings. A computation is either available now (the \text{Now} constructor), or is a computation requiring one unfolding plus some other number of unfoldings (the \text{Later} constructor). The \text{Now} constructors acts as the usual \textit{pure} or \textit{return} operation to inject a pure value into the monad. The monadic flatten can be implemented recursively:

\[
\begin{align*}
\text{flatten} : \mathcal{L} (\mathcal{L} A) & \rightarrow \mathcal{L} A \\
\text{flatten} (\text{Now} x) &= x \\
\text{flatten} (\text{Later} x) &= \text{Later} ((\text{next flatten}) @ x)
\end{align*}
\]

In \text{TCTT}, \text{@} and \text{next} are defined in such a way that the recursive use of \text{flatten} in the \text{Later} case is seen as structurally decreasing. With these operations, we can define a monadic bind in the usual way, and can use Haskell-style do-notation, which greatly improves the readability of our terms.

\subsection{6.2.5.3 Functions}

Functions abstract over computation. Because of this, to model functions in exact mode, we need to capture functions that produce computations, not just pure values. We do so using the lifting monad:

\[
\text{El Exact } (\text{CΠ dom cod}) = (x : \text{El dom}) \rightarrow \mathcal{L} (\text{El} (\text{cod} x))
\]

So functions internalize impure computations into the language of values: a function can be a pure value that, when applied, performs impure (possibly non-terminating) computation. Having functions produce values from the lifting monad lets us cast between \textit{?} and other function types, since we can use \text{Later} and the
monadic bind to apply a guarded value to a function whose result is monadic.

### 6.2.5.4 The Unknown Type

Since ? is a type in ⊲GEQ, our model needs a code corresponding to it, which we call C?. But what should El C? be? For non-dependent types, New and Ahmed [105] use a recursive sum type to model ?, with a variant corresponding to each type former of the static language:

\[
\text{Unk} := \mathbb{N} + (\text{Unk} + \text{Unk}) + (\text{Unk} \rightarrow \text{Unk}) + \ldots
\]

However, tCt does not allow unrestricted recursive types, and the \((\text{Unk} \rightarrow \text{Unk})\) variant is not strictly positive, so there can be no equivalent inductive type.

Instead, we rely on guarded recursion to model ?. As with non-dependent gradual types, we define an inductive type ?Ty ℓ for each level ℓ with a variant for each type former. For the function case, we give the constructor the following type:

\[
\text{Π} : (\triangleright (?Ty \, ℓ) \rightarrow \mathcal{L} (\triangleright (?Ty \, ℓ))) \rightarrow ?Ty \, ℓ
\]

The tyFix operator lets us define a type that is equivalent to this such a guarded inductive type, which we describe fully in Section 6.3. So we can inject arbitrary functions into ?Ty ℓ, so long as they only refer to their arguments under the guarded modality.

### 6.2.5.5 A Guarded Algebra

One advantage of this chapter’s approach over Chapter 3 is that it gives unified way of presenting the approximate and exact semantics. We can take this further by defining an interface for both approximate and exact semantics, and using these abstract operations to describe approximate and exact computations. The operations we need are as follows, modelled after the operations of Guarded Cubical Agda [151].
\[\begin{align*}
\mathcal{\Phi}_{\mathcal{A}} & : \ Type_\mathcal{A} \rightarrow Type_\mathcal{A} \\
\mathcal{\Gamma}_{\mathcal{A}} & : \mathcal{\Phi}_{\mathcal{A}} Type_\mathcal{A} \rightarrow Type_\mathcal{A} \\
\text{next}_{\mathcal{A}} & : A \rightarrow \mathcal{\Phi}_{\mathcal{A}} A \\
\omega \odot_{\mathcal{A}} & : \mathcal{\Phi}_{\mathcal{A}} (A \rightarrow B) \rightarrow \mathcal{\Phi}_{\mathcal{A}} A \rightarrow \mathcal{\Phi}_{\mathcal{A}} B \\
\text{mapNext}_{\mathcal{A}} & : (f : A \rightarrow B) \rightarrow (x : A) \rightarrow (\text{next}_{\mathcal{A}} f) @ (\text{next}_{\mathcal{A}} x) = \mathcal{\Phi}_{\mathcal{A}} B \ \text{next}_{\mathcal{A}} (f \ x) \\
\text{dfix}_{\mathcal{A}} & : (\mathcal{\Phi}_{\mathcal{A}} A \rightarrow A) \rightarrow \mathcal{\Phi}_{\mathcal{A}} A \\
\text{pfix}_{\mathcal{A}} & : (f : \mathcal{\Phi}_{\mathcal{A}} A \rightarrow A) \rightarrow \text{dfix}_{\mathcal{A}} f = \mathcal{\Phi}_{\mathcal{A}} A (\text{next}_{\mathcal{A}} (f (\text{dfix}_{\mathcal{A}} f))) \\
\text{Eq}_{\mathcal{A}} & : (f : \mathcal{\Phi}_{\mathcal{A}} Type_\mathcal{A} \rightarrow Type_\mathcal{A}) \rightarrow \mathcal{\Gamma}_{\mathcal{A}} (\text{dfix}_{\mathcal{A}} f) = Type_\mathcal{A} \mathcal{\Phi}_{\mathcal{A}} (f (\text{dfix}_{\mathcal{A}} f))
\end{align*}\]

When \(\mathcal{A} = \text{Exact}\), these operations are all easily implemented in terms of the primitives of \\(\text{gctt}\\. For \(\mathcal{A} = \text{Approx}\), the key is to set \(\mathcal{\Phi}_{\text{Approx}} T = 1\), which ensures that no approximate computation can rely on a guarded value in a non-trivial way. The remainder of the methods are all easily implemented.

Similarly, we can implement a generic lifting monad:

\[
\text{data } L_\mathcal{A} : Type_\mathcal{A} \rightarrow Type_\mathcal{A} \text{ where} \\
\quad \text{Now} : A \rightarrow L_\mathcal{A} A \\
\quad \text{Later} : (\mathcal{A} = \text{Exact}) \rightarrow \mathcal{\Phi}_{\mathcal{A}} (L_\mathcal{A} A) \rightarrow L_\mathcal{A} A
\]

So in exact mode, we get the lifting monad from above, but for approximate mode, we cannot use \text{Later}, so we get an identity monad. When in approximate mode, the monad has an escape operation, since \text{Now} is the only constructor:

\[
\text{from} L : L_{\text{Approx}} A \rightarrow A
\]

We can use a single monadic definition for both approximate and exact versions of eliminators, without losing the ability to remove an approximate value from the monad when we need to.
6.2.6 Elimination Operations

With our lifting monad in hand, we have the essential tools to model the elimination operations of \( \triangleright \text{GEq} \). \( \text{GEq} \) had three sorts of gradual elimination-specific rules: propagation rules (reducing ?), casting rules, and composition rules, so we define three main elimination functions corresponding to these. We model these operations using functions of the following types:

- \( \text{cast} : (c_1 : C \ell) \to (c_2 : C \ell) \to \text{El } c_1 \to \mathcal{L}_{ae} (\text{El } c_2) \)
- \( \text{comp} : (c : C \ell) \to \text{El } c \to \text{El } c \to \mathcal{L}_{ae}(\text{El } c) \)
- \( \text{codeComp} : (c_1 : C \ell) \to (c_2 : C \ell) \to C \ell \)
- \( ? : (c : C \ell) \to \text{El } c \)

Casting takes two codes, and a value of the first code’s type, and produces a value of the second code’s type. Composition takes a code and two values of the code’s type, and produces another value of that code’s type containing all the information from the input values. Code composition takes two codes and produces the code containing all information from both input codes. Finally, ? takes a code and produces the value denoting ? for that code’s type. Casting and composition produce a monadic result, because they may need to convert a function to or from ?. However, computing ? and the composition of two codes only relies on the approximate normal forms of terms, so they can return pure values.

The elimination operations of \( \triangleright \text{GEq} \) are mutually dependent. As we saw in Chapter 5, casting between equality types requires composing witnesses. Composing dependent function types or inductive types requires telescope composition, which in turn requires casts. Casting an inductive type to ? requires computing the germ, which requires computing ? for the type of indices, but computing ? for equality types involves taking composing equated terms. So casting, composition, and computing ? are all mutually dependent.
6.3 Modelling The Unknown, Exactly and Approximately

We begin the presentation of our model with our representation of \(?\), the heart of gradual typing. In this section, we describe the inductive type that captures values of type \(?\). We pay particular attention to how to model inductive types, since they are what could not be modelled in Chapter 3.

6.3.1 Heads

We use a target type to model head tags. We use these to compute germ types, and to compare codes or terms during casting or composition.

```
data Head where
  H0 : Head
  H1 : Head
  HType : Head
  HΠ : Head
  HΣ : Head
  H= : Head
  Hctor : CName → Head
```

These correspond fairly directly to the heads defined in Sections 4.2 and 5.3.1. The main additions are the heads for empty, unit, and pair types. While these can be defined as inductive types, having them as primitives helps us generically represent inductive types.

6.3.2 An Inductive Representation of the Unknown Type

We represent the unknown type and germ types for each head as an inductive family indexed by type Maybe Head. An index of nothing indicates a value of type \(?\), whereas a value of type just \(h\) indicates a member of the germ for \(h\). Represent-
ing these as different types in the same indexed family simplifies the termination argument when traversing values of the unknown type. We also assume we have constructed the model for all levels lower than the current one, and that we can refer to $\mathbb{C} - 1 : \text{Type}$, the code for the previous Universe à la Tarski. In Section 6.4.3 we tie the knot by defining our codes by recursion on universe levels.

```
data $\text{Germ}_{\ell : \mathbb{N}} \{ \ell : \mathbb{N} \} : \text{Maybe} \text{ Head} \rightarrow \text{Type} \text{ where}$
  $\text{??} : ?\text{Ty}$
  $\mathcal{U} : ?\text{Ty}$
  $\text{?Tag} : (h : \text{Head}) \rightarrow ?\text{Germ} (\text{just} \ h) \rightarrow ?\text{Ty}$
  $\text{?=} : ?\text{Germ} (\text{just} \ H\text{=})$
  $\text{Type} : (\ell > 0) \rightarrow \mathbb{C} - 1 \rightarrow ?\text{Germ} (\text{just} \ H\text{Type})$
  $\text{?}\Pi : (f : >_{\text{ax}} (?\text{Ty}) \rightarrow (\mathcal{L}_{\text{ax}} (?\text{Ty}))) \rightarrow (\text{limit} : ?\text{Ty}_{\text{Approx}}) \rightarrow ?\text{Germ} (\text{just} \ H\Pi)$
  $\text{Ctor} : (\text{tyctor} : \text{CName}) \rightarrow (d : \text{DName} \ \text{tyctor})$
    $\rightarrow ((r : \text{GermResponse} (\text{germCtor} \ \ell \ \text{tyctor} \ d)) \rightarrow \mathcal{L}_{\text{ax}} (?\text{Ty} \ \ell))$
    $\rightarrow ((r : \text{GermResponse} (\text{germCtor} \ \ell \ \text{tyctor} \ d)) \rightarrow ?\text{Ty}_{\text{Approx}} \ \ell)$
  $\rightarrow ?\text{Germ} (\text{just} \ H\text{Ctor} \ \text{tyctor}))$

$?\text{Ty}_{\text{ax}} = ?\text{Germ}_{\text{ax}} \text{ nothing}$
```

For clarity, we write this type as if we had declared it directly as an inductive type that was allowed to refer to itself negatively under the guarded modality. Taking its fixed point with $\text{tyfix}$ produces an equivalent type, but adds significant notational overhead. An element of $?\text{Ty}$ is an element of $?\text{Germ nothi}ng$, meaning that it denotes $?$ or $\mathcal{U}$, or is a tagged value of $?\text{Germ (just h)}$ for some head $h$. An element $?\text{=} of the unit germ has no data, and an element of the pair germ contains two values of $?\text{Ty}$. An element of the germ for equality is another member.
of $Ty$, which serves as a GEQ-style witness for the equality between $?$ and itself. A type from a lower universe can be injected into the germ with $Type$. Embedding values from a lower level is lossless: just as Lennon-Bertrand et al. [87] use universe restrictions on functions to make GCI strongly normalizing, we can directly inject functions from lower-level universes into a higher germ without approximation or guardedness.

The constructor $\Pi$ for the function germ is the most subtle. It has two fields. The first is a function from $L_\alpha L_\alpha$ to $\ell_\alpha L_\alpha$. Having the domain under the guarded modality ensures that the type is still strictly positive, and having the return under $L_\alpha$ ensures that we can actually produce a result that uses the argument, as well as allowing possibly non-terminating computations to be represented. The second field is a limit, which is used in the exact version of $\Pi$ to produce an approximation of the function that produces the same output for all inputs. The limit is necessary because we may need to refer to the function in a type computation, but the function itself returns a value under the $L_\alpha$ monad, and determining if the described computation results in a pure value is equivalent to solving the halting problem. When we define casts in Section 6.5, we ensure that the limit is always instantiated to the function being cast, applied to $?$. We explain $\text{Ctor}$ in more detail in Section 6.3.3.1, but for now it suffices to know that $\text{CName}$ is the type of type constructors and $\text{DName tyctor}$ is the type of data constructors for a given type constructor. A member of the germ for an inductive type contains a data constructor for that type, and a mapping from the germ response type for that constructor to elements of $Ty$. The germ response type is modelled after responses from W-types (Section 2.2.3.2), and is essentially a record that describes which field of the constructor is being accessed. Like with functions, we need to store an approximate version outside the $L_\alpha$ monad that we can use in type computations. Unlike with the germ for functions, because the domain is not $Ty$, we can store an approximate value for every input, as opposed to a single limit over all inputs.
6.3.3 The Germ for Inductive Types

One of the major challenges in constructing our model is handling inductive types generically. Our approach must be flexible enough to give confidence that we can model any user-defined inductive types, but structured enough to fit within the confines of our model. Moreover, we need to build the infrastructure for inductives piece by piece. For example, we cannot describe the fields of a constructor in terms of codes until we have defined codes.

The germ for an inductive type has a tricky mutual dependency with the unknown type. We need to traverse the members of inductive types that are embedded into Ty, in order to cast them and compose them with each other. However, fields of those inductives may themselves be of the unknown type. Both traversals need to be done in a way that the TCTT type system recognizes as terminating.

Our strategy is to define the germ for inductive types mutually with the unknown type. More specifically, we define it in the same inductive family Germ with a different index. So the usual eliminator for an inductive family can be used to describe traversing both members of Ty and germ types. We use a version of inductive descriptions to describe the field structure of a data constructor in the germ without needing to have the germ or Ty defined beforehand. We then use a version of W-types to incorporate these descriptions into the inductive structure of Germ.

6.3.3.1 Signatures and Constructor Tags

The first piece of any inductive type is what we call a signature, which is a linked-list structure that specifies how many fields a data constructor has, and whether each field is merely an argument (not recursive) or an element of the type itself. For argument and recursive fields, we store their arity, a lower-bound on the number of arrows in the type. The signature is then used to index descriptions of inductive types, so that we can traverse different types in parallel if they have the same signature (for example, if they are instantiations of the same parameterized
We parameterize the entire model over an arbitrary but fixed set of inductive types. Nearly every module of our development takes such a record as a parameter, describing what inductive types are declared, what their constructors are, and a signature describing the shape of the fields for each constructor.

\[
\text{data IndSig where}
\begin{align*}
\text{EndE} & : \text{IndSig} \\
\text{ArgE} & : (\text{arity} : \mathbb{N}) \rightarrow \text{IndSig} \rightarrow \text{IndSig} \\
\text{RecE} & : (\text{arity} : \mathbb{N}) \rightarrow \text{IndSig} \rightarrow \text{IndSig}
\end{align*}
\]

We use \( \text{CName} := \text{Fin numTypes} \) and \( \text{DName} := \lambda \text{tyctor} \rightarrow \text{Fin (numCtors tyctor)} \) as shorthand for the types of type constructors and data constructors respectively.

### 6.3.3.2 Descriptions for Germ Constructors

What is in the germ for each inductive type? Above we saw that \(?\text{Ty}\) can contain elements of the germ for each constructor. In \text{GEQ} the germ was the inductive type with \(?\) supplied for each parameter, but in our model we have not yet defined codes describing types, let alone a value \(?\) for each code’s type. Moreover, replacing parameters with \(?\) may violate strict positivity. One could define a type \( \text{C (A : Type) (B : Type) : Type} \) with a single constructor \( \text{D : (A \rightarrow B) \rightarrow C A B} \). But if the germ is \( \text{C ???} \) and \( \text{Typer} \) embeds this germ, in the model \( \text{?Ty} \) would contain unguarded functions from \( \text{?Ty} \) to itself.

Instead, we store each field by casting to \(?\) or a function producing \(?\). The structure is given by a description for each germ constructor, which is a simplified
version of an inductive description (Section 2.2.3.3). These descriptions give us a way to describe how the germ of an inductive type looks, in terms of fields with type $\mathcal{Ty}$, before we have actually defined $\mathcal{Ty}$, which ensures that we can only use it strictly positively.

```haskell
data GermTele : ℕ → Type, where
  GNil : GermTele 0
  GCons : (A : Type) → (A → GermTele n) → GermTele(S n)
```

```haskell
data GermCtor : IndSig → Type where
  GEnd : GermCtor SigE
  GArg : {n : ℕ} {sig : IndSig}
       → (dom : GermTele n) → (D : GermCtor sig) → GermCtor (SigA n sig)
  GRec : {n : ℕ} {sig : IndSig}
        → (dom : GermTele n) → (D : GermCtor sig) → GermCtor (SigA n sig)
```

$\text{GEnd}$ marks that there are no further constructors. $\text{GArg}$ and $\text{GRec}$ have identical field types, because the germ encodes both recursive and non-recursive fields of inductives using $\mathcal{Ty}$. Having separate argument and recursive constructors is useful when defining casts from inductive types to the germ (Section 6.5.5.3). For a signature with arity $n$, $\text{GArg}$ and $\text{GRec}$ each take a telescope of length $n$, which is just $n$ nested dependent pair types. If the field has type $(x_1 : T_1) \rightarrow \cdots (x_n : T_n) \rightarrow T$, then the domain telescope is $(x_1 : T_1) \times \cdots \times T_n$.

Our definition of $\mathcal{Germ}$ is parameterized over a function:

```haskell
germcutor : (tytor : CName) → (d : DName tyctor) → GermCtor (indSig tyctor d)
```

As we add more machinery to our model, we will add more parameters refining the structure of inductive types.

The $\text{GermCtor}$ type describes the type of the germ for a particular type and data constructor, but to actually store things of those types, we interpret them
into the germ response types that $\texttt{\textit{Ctor}}$ referred to:

\begin{align*}
\text{GermTeleEnv} & : \text{GermTele } n \to \text{Type} \\
\text{GermTeleEnv} \text{ GNil} & = \text{1} \\
\text{GermTeleEnv} \text{ (GCons } A \text{ rest}) & = (x : A) \times \text{GermTeleEnv} \text{ (rest } x) \\
\text{GermResponse} & : \text{GermCtor } \text{sig} \to \text{Type} \\
\text{GermResponse} \text{ GEnd} & = \text{0} \\
\text{GermResponse} \text{ (GArg } \text{ dom } D) & = \text{GermTeleEnv} \text{ dom } + \text{GermResponse} \text{ D} \\
\text{GermResponse} \text{ (GRec } \text{ dom } D) & = \text{GermTeleEnv} \text{ dom } + \text{GermResponse} \text{ D}
\end{align*}

Then the $\texttt{\textit{Ctor}}$ constructor interprets the description for a germ constructor into a simplified version of a $W$-type. Recall its type:

\begin{align*}
\texttt{\textit{Ctor}} & : (\texttt{tyctor} : \texttt{CName}) \to (d : \texttt{DName tyctor}) \\
& \to ((r : \texttt{GermResponse} (\texttt{germCtor } \ell \texttt{ tyctor } d)) \to L_{\approx} (\texttt{Ty } \ell)) \\
& \to ((r : \texttt{GermResponse} (\texttt{germCtor } \ell \texttt{ tyctor } d)) \to ?\texttt{Ty}_{\texttt{Approx}} \ell) \\
& \to ?\texttt{Germ} \texttt{(just } (\texttt{HCtor } \texttt{tyctor}))
\end{align*}

So the germ contains an uncurried version of a function that, for each argument, yields a function producing $\texttt{?}$. In our example above, the field with type $(x_1 : T_1) \to \ldots (x_n : T_n) \to T$ is modelled as an uncurried function from the corresponding telescope $((x_1 : T_1) \times \ldots (T_n)) \to ?\texttt{Ty } \ell$.

### 6.3.3.3 Example: Vectors

Consider the inductive type of vectors from Section 2.2.2.6, with equality constraints instead of indices:

\begin{verbatim}
data Vec (A : Type)(n : \texttt{N}) : Type where
  Nil : (n = 0) \to Vec A n
  Cons : (m : \texttt{N}) \to A \to \text{Vec A m} \to (n = S m) \to \text{Vec A n}
\end{verbatim}
The signatures for \textbf{Nil} and \textbf{Cons} would be:

\begin{verbatim}
nilSig : SigA 0 SigE
consSig : SigA 0 (SigA 0 (SigR 0 (SigA 0 SigE)))
\end{verbatim}

The constructors would be:

\begin{verbatim}
GArg GNil GEnd : GermCtor nilSig
GArg GNil (GArg GNil
  (GRec GNil (GArg GrmTeleNil GEnd))) : GermCtor consSig
\end{verbatim}

The responses for these are respectively:

\begin{verbatim}
1 + 0
1 + 1 + 1 + 1 + 0
\end{verbatim}

All of this is a long-winded way to say that for \textbf{Nil}, our model’s germ for \textbf{Vec} contains one value of type \texttt{?Ty}, storing the witness of the equality between \texttt{n} and \texttt{0}. For \textbf{Cons} it has four values of type \texttt{?Ty}: the tail length, the head, the tail, and the witness that the list length is one longer than the tail.

If, for example, we designed a type \texttt{SequenceVec}, where \textbf{Cons} took a head of type \texttt{N \to A} instead of just \texttt{A}, then the signature for \textbf{Cons} would be:

\begin{verbatim}
consSig : SigA 0 (SigA 1 (SigR 0 (SigA 0 SigE)))
\end{verbatim}

The second \texttt{GNil} in the constructor is replaced with \texttt{GCons N (\lambda x. GNil)}, and the response type for \textbf{Cons} would be:

\begin{verbatim}
1 + N + 1 + 1 + 0
\end{verbatim}

\textbf{6.3.3.4 Approximation for Inductive Types}

Our approach to inductives causes approximations to happen. By defining all fields of inductive germ types to be \texttt{?Ty} or function types with codomain \texttt{?Ty}, we
obtain a straightforward way to traverse them within the confines of our target

type theory. However, with approximate normalization, converting to $?\text{Ty}$ is a

lossy operation.

In approximate mode, constructor fields whose contents contain functions

may be approximated. We try to reduce this approximation as much as possible

by only converting the results of functions to $?\text{Ty}$. (We explain how this interacts

with our termination argument in Section 6.5). However, if a field’s type contains

$?$ to the left of a function arrow in the germ, then in our model, that field must be

given arity 0 so that it can be approximated. This approximation is by design and

is a critical part of our termination argument.

However, some unnecessary approximations may happen on functions that

are nested deeper within data types. If a constructor field has type $\text{List} (N \rightarrow N)$,

when a value for that field is converted to $?\text{Ty}$, all the functions in the list are con-

verted to $1 \rightarrow ?\text{Ty}$, even though we could safely embed a function of type $N \rightarrow ?\text{Ty}$

into $?\text{Ty}$. Even in exact mode, these overapproximations can happen, because wit-

nesses of equality are stored in approximate form.

We believe that future work will be able to remove these unnecessary approxi-

mations. The main impediment to removing them now is extra restrictions placed

on termination in Cubical Agda [63]. Since we do not use univalent features, these

restrictions can probably be soundly removed, but for this dissertation, we favour

over-approximation over possibly compromising the soundness of the termina-

tion argument.

6.4 Modelling Types: A Gradual Universe of Codes

We now have a type to model $?$, but $?$ is only one type in $\triangleright \text{GEQ}$. In this section, we

show how all the types of $\triangleright \text{GEQ}$ are modelled, using a Universe à la Tarski, with

codes that are interpreted to types.

6.4.1 Gradual W-Types

In this section, we use a modified version of W-types from Section 2.2.3.2, defined

as follows:
data $W (S : Type) (P : S \rightarrow L \wedge Type) (P_{\text{approx}} : S \rightarrow Type)$ where

$W^{\text{sup}} : \lambda x : S . \rightarrow (P x \rightarrow L (W S P P_{\text{approx}})))$

$W^{\text{f}} : \lambda x : S . \rightarrow (W S P P_{\text{approx}})$

$W^{?} : \lambda x : S . \rightarrow (P_{\text{approx}} x \rightarrow (W S P P_{\text{approx}}))$

We add two constructors, $W^{?}$ and $W^{\text{f}}$, which correspond to $?_{\text{c}}$ and $U_{\text{c}}$ for an inductive type $C$. We also modify the definition of $\text{Container}$ to store two functions, one producing a result under $L_{\wedge}$ and one producing a pure approximation. This enables $\text{approx}$ to be implemented in the next section.

### 6.4.2 Codes for inductive types

The universe of codes is defined as a giant, mutually inductive-recursive definition, with several helper functions and types.

\[
\begin{align*}
C : & \mathbb{N} \rightarrow Type \\
E_{\wedge} : & C \ell \rightarrow Type \\
\text{approx} : & (c : C \ell) \rightarrow E_{\wedge} c \rightarrow E_{\text{approx}} c \\
\text{exact} : & (c : C \ell) \rightarrow E_{\text{approx}} c \rightarrow E_{\wedge} c \\
\text{approxExact} : & (c : C \ell) \rightarrow (x : E_{\text{approx}} c) \rightarrow \text{approx} (\text{exact} x) = x \\
\end{align*}
\]

\[
\begin{align*}
\text{CDesc} : & (c_{\text{Arg}} : C \ell) \rightarrow \text{IndSig} \rightarrow Type \\
\text{interpDesc} : & \text{CDesc} c_{\text{Arg}} \rightarrow E_{\text{approx}} c_{\text{Arg}} \rightarrow \text{Container}
\end{align*}
\]

For clarity, we present each code along with its interpretation, instead of as two monolithic definitions. We omit the definitions of $\text{approx}$, $\text{exact}$, and $\text{approxExact}$,
since they are conceptually straightforward but very long.

### 6.4.2.1 Unknown and Error

\[
\begin{align*}
C? &: C \ell \\
El_{ae} C? &= ?Ty_{ae} \ell \\

CU &: C \ell \\
El_{ae} CU &= 1
\end{align*}
\]

The codes denoting ?Type, and ?Type, contain no arguments. The unknown type is interpreted with our type from the previous section. The error type is interpreted to the unit type, since its only value is a dynamic type error. So ?Type is equal to ?Type.

### 6.4.2.2 Unit and Empty

\[
\begin{align*}
C1 &: C \ell \\
El_{ae} C1 &= B \\

C0 &: C \ell \\
El_{ae} C0 &= 1
\end{align*}
\]

The codes for unit and empty have no arguments. The interpretation of the unit type has two elements: one denoting unit : Unit and one denoting an error. So ?Unit is equal to unit : Unit. For the empty type, the interpretation is identical to the error type. However, we keep these as separate codes because they are treated differently by precision and consistency. If CU shows up in a type, then an error has occurred when computing that type, but C0 denotes a perfectly reasonable static type and should not indicate an error. An inhabitant of either type must be an error.
6.4.2.3 Functions

\[ C\Pi : (\text{dom} : C \ell) \rightarrow (\text{cod} : \text{El}_{\text{Approx}} \text{dom} \rightarrow C \ell) \rightarrow C \ell \]

\[ \text{El}_\approx (C\Pi \text{dom cod}) = \]

\[ (x : \text{El dom}) \rightarrow (L_\approx (\text{El}_\approx (\text{cod} (\text{approx} x)))) \times (\text{El}_{\text{Approx}} (\text{cod} (\text{approx} x))) \]

With functions, we start to see how approximation fits into the design of our model. The code for a function takes a code for its domain type and a codomain code for each approximate element of the domain, so that we can encode dependent function types. Functions (and other value-dependent types) are why we need induction-recursion: we cannot define the type of the code for a function without the interpretation of its domain code.

For the interpretation of functions, we see the same pattern that we saw with \(?\text{Ty}\) and inductive germs. A function code is interpreted to a function that, for every input, produces a pair containing an exact computation under \(L_\approx\) and a pure approximation. The result type of the function is computed using the approximation of the argument’s value, avoiding the need to handle non-termination in type computations.

6.4.2.4 Pairs

\[ C\Sigma : (\text{dom} : C \ell) \rightarrow (\text{cod} : \text{El}_{\text{Approx}} \text{dom} \rightarrow C \ell) \rightarrow C \ell \]

\[ \text{El}_\approx (C\Sigma \text{dom cod}) = \]

\[ (x : \text{El dom}) \times (\text{El}_{\text{Approx}} (\text{cod} (\text{approx} x))) \]

Dependent pairs work like functions, but without the need to put anything behind the \(L_\approx\) monad. The type of the code constructor is exactly the same: we have a domain code describing the type for the first element, and a codomain code for each approximate value of the first element. The interpretation is a dependent pair type, where the second element’s type is computed using the approximation of the first element’s value.
6.4.2.5 Types

\[
\text{ CType : } (\ell > 0) \rightarrow C \ell \\
\text{ El}_\ell \text{ CType } = C - 1
\]

If we are constructing the universe of codes at a level higher than zero, then we can include both codes and values from lower universes. The CType constructor models Type$_{\ell-1}$, and is interpreted to be the type of all codes from one universe level lower. In Section 6.4.3 we describe how we compute the lower-level codes recursively.
6.4.2.6 Equality

Equality is modelled in a straightforward adaptation of GEQ’s equality, with one major change: witnesses are approximate. The code constructor takes a code for the type of elements being compared, and two approximate elements of that code’s type, which are the equated values. The interpretation of an equality code is the approximation of the type of the equated values: an inhabitant of the gradual equality between terms is a witness of their consistency.

Having the equated terms in the code be approximate is necessary for decidable type checking, but having the witness be approximate is a design choice. We could change the interpretation to be \( \text{El}_{\approx} c \) and for the most part the language would be unchanged. However, having approximate witnesses helps with the usability of the language: since witnesses are not in GEQ’s surface language (and hence, not in a hypothetical >GEQ surface language), their computations should be opaque to the programmer. If a witness computation does not terminate, then finding the source of the error is extremely difficult for the programmer, because the loop is happening in code they did not write and cannot see.

6.4.2.7 Inductive Types

As with the germ types, handling inductive types introduces fairly significant complexity into our universe of codes. To generically describe inductive types in terms of codes, we use a technique originated by Diehl and Sheard [49]. We define an inductive type of descriptions, specifying inductive types whose fields are themselves denoted by codes. We then interpret those “closed-universe” descriptions to an open-universe description, in our case, a gradual W-type.

First, a helper predicate \( \text{HasArityAtLeast} \) ensures a given nested function or
pair type has the required depth:

\[
\text{data HasArityAtLeast : Head} \to \mathbb{N} \to C \ell \to \text{Type where}
\]
\[
\begin{align*}
\text{Arity0 : } & (c : C \ell) \to \text{HasArityAtLeast } h 0 c \\
\text{ArityΠ} : & (\text{dom} : C \ell) \to (\text{cod} : \text{ElApprox} \text{ dom} \to C \ell) \\
& \to ((x : \text{ElApprox} \text{ dom}) \to \text{HasArityAtLeast } H\Pi n (\text{cod} x)) \to \text{HasArityAtLeast } H\Pi (S n) (\text{CΠ dom cod}) \\
\text{ArityΣ} : (\text{dom} : C \ell) \to (\text{cod} : \text{ElApprox} \text{ dom} \to C \ell) \\
& \to ((x : \text{ElApprox} \text{ dom}) \to \text{HasArityAtLeast } H\Sigma n (\text{cod} x)) \\
& \to \text{HasArityAtLeast } H\Sigma (S n) (\text{CΣ dom cod})
\end{align*}
\]

Then, we have a type of descriptions over a given signature.

\[
\text{data CDesc : } (c_{\text{Arg}} : C \ell) \to \text{IndSig} \to \text{Type where}
\]
\[
\begin{align*}
\text{CEnd} : & C\text{Desc } c_{\text{Arg}} \text{ SigE} \\
\text{CArg} : (c_{\text{Field}} : \text{ElApprox } c_{\text{Arg}} \to C \ell) \\
& \to ((\text{arg} : \text{ElApprox } c_{\text{Arg}}) \to \text{HasArityAtLeast } H\Pi n (c_{\text{field}} \text{ arg})) \\
& \to (D : C\text{Desc } (C\Sigma c_{\text{Arg}} c_{\text{Field}}) \text{ sigRest}) \\
& \to C\text{Desc } c_{\text{Arg}} (\text{SigA n sigRest}) \\
\text{CRec} : (\text{dom} : C \ell) \\
& \to ((\text{arg} : \text{ElApprox } c_{\text{Arg}}) \to \text{HasArityAtLeast } H\Sigma n (\text{dom arg})) \\
& \to (D : C\text{Desc } c_{\text{Arg}} \text{ sigRest}) \\
& \to C\text{Desc } c_{\text{Arg}} (\text{SigR n sigRest})
\end{align*}
\]

The \(c_{\text{Arg}}\) index captures the types from previous fields on which this field can depend. Though not typically used in inductive descriptions, they enable us to ensure that even if field types depend on previous argument values, the signature of an inductive type is statically known. Every description starts with an index of \(C1\), but builds the index up in deeper fields.

The \(\text{CEnd}\) constructor denotes no more fields. The \(\text{CArg}\) constructor describes a non-recursive field with argument type given by interpreting \(c_{\text{Field}}\), which can
depend on the values of previous fields. We take a predicate to ensure that the
type is at least \( n \) nested function types, which lets us avoid unnecessarily ap-
proximating the first \( n \) layers of functions when converting to \( ?T_y \). Finally, the
description contains a recursive field describing the rest of the constructor, with
this field type added to the argument type, because later field types may depend
on the value of this field. The \( C_{\text{Rec}} \) constructor describes a recursive field. If we
are constructing datatype \( C \), then the domain code for \( C_{\text{Rec}} \) denotes the type \( T \)
for which this field has type \( T \to C \). Again we ensure that the domain has the
given arity, although we uncurry and ensure that it is \( n \) nested pair types, since
the function’s final domain type is the recursive reference, and is not given by the
code. We then have a description giving the remainder of the fields.

Consider our vector type from Section 6.3.3.3. Assuming codes \( C_A \) and \( C_N \)
describing the vector’s content type and natural numbers respectively, the de-
scriptions for nil and cons of type \( \text{Vec} (\text{El} A) \) \( n \) would be:

\[
\begin{align*}
C_{\text{Arg}} (\lambda_-, C= n \ 0) : & \ C_{\text{Desc}} C \bot \ \text{nilSig} \\
C_{\text{Arg}} (\lambda, C n) & \ \text{Arity0} ( \\
C_{\text{Arg}} (\lambda_-, c A) & \ \text{Arity0} ( \\
C_{\text{Rec}} (\lambda_-, \bot) & \ \text{Arity0} ( \\
C_{\text{Arg}} (\lambda_-(m).C= c n (S m) \ n) & \ \text{Arity0 \ CEnd})) \\
& : \ C_{\text{Desc}} C \bot \ \text{consSig}
\end{align*}
\]

Like with the germ, \( \text{Nil} \) is modelled with a single field that stores an equality
witness between the length and zero. There are four fields for \( \text{Cons} \): the length
of the tail, the head, the tail, and the witness between the length and one plus the
tail length.

The function \( \text{interpDesc} \) interprets descriptions into a command and response
(Section 2.2.3.2), which make up the container whose fixed-points we take using
gradual \( W \)-types. The command and response components are computed as fol-
lows:

\[
\text{CommandD : } \mathbb{C} \text{Desc } c_{\text{Arg}} \text{ sig } \to \text{ApproxEl } c_{\text{Arg}} \to \text{Type}
\]

\[
\text{CommandD CEnd arg } = 1
\]

\[
\text{CommandD (CArg } c_{\text{Field - D}) arg } = \text{ (x : El (c_{\text{Field arg}) } \times \text{ (CommandD D (arg, approx x)))}
\]

\[
\text{CommandD (CRec } _{-D}) arg = \text{ CommandD D arg}
\]

\[
\text{ResponseD : } \mathbb{D} : \mathbb{C} \text{Desc } c_{\text{Arg}} \text{ sig } \to \text{ (arg : ApproxEl } c_{\text{Arg})} \\
\to \text{ (CommandD D arg) \to Type}
\]

\[
\text{ResponseD CEnd arg com } = \emptyset
\]

\[
\text{ResponseD (CArg _{-}) arg com } = \text{ ResponseD arg com}
\]

\[
\text{ResponseD(CRec dom - D) arg } = (\text{El (dom arg)}) + (\text{ResponseD D arg})
\]

\[
\text{mergeContainers : (DName tyctor } \to \text{ Container) } \to \text{ Container}
\]

\[
\text{mergeContainers } f = ((d : \text{DName tyCtor}) \times \text{fst (f d), } \lambda \text{com. snd (f (fst com))}
\]

For \text{CArg}, we store data in the command and just calculate the response recursively, and for \text{CRec}, we calculate the response recursively and use the domain type as the response, so our final field has a function from the domain type to the inductive type. The \text{mergeContainers} helper function turns a container for each data constructor into one container with a field containing a constructor tag followed by the fields for that constructor.
Finally, we have what we need to define the codes for inductive types:

\[
\begin{align*}
C_\mu & : (\text{tyctor : } C\text{Name}) \\
& \quad \to (c_1 : C \ell) \\
& \quad \to (Ds : (d : D\text{Name tyCtor}) \to C\text{Desc }C 1 \text{ (indSkeleton tyCtor d)}) \\
& \quad \to \text{ApproxEl } c_1 \\
& \quad \to C \ell \\
\text{El}_\text{eq} \ (C_\mu \text{ tyctor } c_1 \ Ds \ i) = \\
& \quad W (\text{interpDesc (mergeContainers}(\lambda d. \text{interpDesc (Ds d) true}))
\end{align*}
\]

The code constructor for inductive types takes a type constructor and an index type for that constructor. There is a map \( Ds \) containing a description for each data constructor, and a value for the index. In principle, the code of the index type for a given type constructor should be fixed, or at least fixed for a given set of parameters. However, we cannot define the codes for each constructor’s index type until we have finished defining codes, so we pass the index type in explicitly. The index value is not used, apart from when computing the meet of two codes, but we anticipate it being useful for future work that extends the metatheory of this model.

The interpretation of an inductive code type is the fixed point of the container that contains all of its constructors’ containers. Because \( W \) is strictly positive in its parameter, such a fixed point is allowed in our target type theory.

### 6.4.3 Tying the Knot

Finally, we have a code for each \( \text{GEq} \) type former, and a type corresponding to elements of that type. The last detail is to fill in the references to \( C_{-1} \). We can do this inductively: we define \( C_0 \) with \( C_{-1} := 0 \). Defining it this way does not cause issues because any references to \( C_{-1} \) were hidden behind a proof that \( \ell > 0 \), so when \( \ell \) is 0, the constructors that references \( C_{-1} \) cannot be used. In the recursive case, for \( C \ (\ell + 1) \), we can set \( C_{-1} \) to \( C \ell \).
6.5 Elimination Operations on Values

In this section we describe how we model the $\triangleright$GEq elimination forms. Due to the way we establish termination, the actual terms for these operations are unfortunately quite long. That said, the implementations are often straightforward adaptations of the reduction rules, including those inherited from GEq or GCIC. We attempt to strike a balance between providing the details of how the model is actually constructed, and omitting detail so that the core intuition can be provided. As such, in some parts we give a high-level prose description of the terms, or write them in pseudocode rather than the full detail of the theory. As always, the masochistic curious reader can see the full Agda code in [56].

6.5.1 Operations for Inductives

To define our operations, we parameterize over one more property for inductives. The codes for inductive types contained descriptions, which were themselves built out of codes. However, the descriptions for germs were not, since they were defined before codes. These definitions are parameterized over a predicate which gives, for each germ constructor, a code for each type in its domain telescope, along with proofs that the code’s element type is isomorphic to the domain type. So we can cast to and from code types when managing the arguments to functions in the germ of an inductive type.

That said, we do not write out the terms for casting and composition of inductive types. Because of W-type approach to inductives, casting and composition for inductives is effectively a combination of the same operations on dependent functions and pairs. The functions for dealing with W-types are nigh incomprehensible, so we choose instead to describe the interesting parts of the implementation.

6.5.2 Dynamic Errors

Each code has an operation to compute an error at its type. It is computed using straightforward recursion, and because it does not rely on composition or casting,
it does not need to be part of our mutual recursion.

\[ \text{UFor} : (c : C \ell) \to \text{El}_c c \]
\[ \text{UFor}_c C ? = \emptyset \]
\[ \text{UFor}_c C U = \text{unit} \]
\[ \text{UFor}_c C 0 = \text{unit} \]
\[ \text{UFor}_c C 1 = \text{false} \]
\[ \text{UFor}_c C \text{Type} = C U \]
\[ \text{UFor}_c (C \prod \text{dom} \text{cod}) = \]
\[ \text{pure} (\text{UFor}_c (\text{cod} (\text{UFor}_c \text{dom})))) \]
\[ \lambda x. (\text{UFor}_{\text{Approx}} (\text{cod} (\text{UFor}_{\text{Approx}} \text{dom}))) \]
\[ \text{UFor}_c (C \Sigma \text{dom} \text{cod}) = (\text{UFor}_c \text{dom}, \text{UFor}_c (\text{cod} (\text{UFor}_{\text{Approx}} \text{dom}))) \]
\[ \text{UFor}_c (C = c x y) = \text{Ufor} c \]
\[ \text{UFor}_c (C \mu \text{tyctor} c D x) = \text{WU} \]

We use this as a result when composing terms or casting between codes whose heads are mismatched.

### 6.5.3 Well-Founded Recursion on Codes and Values

Modelling the sizes of \( \ast \text{GEq} \) types requires some subtle tricks, which we outline here. When casting between types or composing two types, we need sizes for two codes. The symmetry of both of these operations means that we cannot compare the sizes using a lexicographic ordering, since neither argument’s size can dominate the other. We also need to use ordinals, rather than the depth of the trees, because we need the code size for \( (x : T_1) \to T_2 \) to be strictly larger than \( T_2 \) instantiated with any value for \( x \). Finally, we need our ordinals to be constructive, so we can express them in \texttt{tctt}.

To see why we cannot use a lexicographic ordering, consider the composition:

\[(x : T_1) \to T_2 \& \text{Type} (x : T_1') \to T_2'\]

We compose the domains to get \( T_1 \& \text{Type} T_1' \). However, to compose the codomains,
we need to cast the argument \( x \) from \( T_1 \&_{\text{Type}} T'_1 \) to both \( T_1 \) and \( T'_1 \). So we need both of the following inequalities to hold:

\[
\begin{align*}
(T_1 \&_{\text{Type}} T'_1, T_1) &< (x : T_1) \rightarrow T_2, (x : T'_1) \rightarrow T'_2 \\
(T_1 \&_{\text{Type}} T'_1, T'_1) &< (x : T_1) \rightarrow T_2, (x : T'_1) \rightarrow T'_2
\end{align*}
\]

This inequality does not hold in general for a lexicographical order: there is no reason that \( T'_1 \) cannot be massively larger than both \( T_1 \) and \( T_2 \).

Instead, we compare pairs of types by defining a maximum operation \( \lor \) on the ordinals describing their sizes. In addition to being a true least upper-bound, we also need \( \lor \) to be strictly monotone, that is, to satisfy \( s_1 \lor s_2 < s'_1 \lor s'_2 \) whenever \( s_1 < s'_1 \) and \( s_2 < s'_2 \). This inequality arises because each recursive call must be made on a strictly smaller argument. Finally, we show that the size of the composition of two types is bounded by the maximum of the size of either types. Given these properties, we can properly order the sizes in the first inequality above:

\[
\begin{align*}
(T_1 \&_{\text{Type}} T'_1 \lor T_1) &< (x : T_1) \rightarrow T_2 \lor (x : T'_1) \rightarrow T'_2 \\
&< (x : T_1) \rightarrow T_2 \lor (x : T'_1) \rightarrow T'_2 \\
&< (x : T_1) \rightarrow T_2 \lor (x : T'_1) \rightarrow T'_2
\end{align*}
\]

6.5.3.1 Brouwer Trees

Unfortunately, it was not immediately apparent that any of the “off-the-shelf” formulations of constructive ordinals satisfied our criteria, so we built our own
formulation. We use a refined version of Brouwer trees:

data Ord : Type where
    OZ : Ord
    O↑ : Ord → Ord
    OLim : (c : C) → (ElApprox c → Ord) → Ord

There is a zero ordinal, a successor operator, and a limit ordinal that is the least upper bound of the image for a function from a code’s type to ordinals. We borrow the trick of taking the limits over types (or in our case, codes) from Chan [27], since this lets us easily model the sizes of dependent functions and pairs. The ordering on these trees is defined following Kraus et al. [83]:

data _≤_ : Ord → Ord → Type where
    _≤_Z : (o : Ord) → OZ _≤_ o
    _≤_SucMono : (o₁ : Ord) → (o₂ : Ord) → o₁ _≤_ o₂ → O↑ o₁ _≤_ o₂
    _≤_cocone : (c : C) → (o : Ord) → (f : ElApprox c → Ord) → (k : ElApprox c)
        → o _≤_ o f k → o _≤_ OLim c f
    _≤_limiting : (o : Ord) → (c : C) → (f : ElApprox c → Ord)
        → (k : ElApprox c) → f k _≤_ o → OLim c f _≤_ o

    o₁ <_o o₂ = O↑ o₁ _≤_ o₂

That is, zero is the smallest ordinal, the successor is monotone, and the limit is actually the least upper bound of the function’s image. Unlike Kraus et al. [83], we do not include transitivity as a rule, but we can prove it as a theorem. The
maximum function on ordinals is defined as follows:

\[
\begin{align*}
\text{max}_0 & : \text{Ord} \to \text{Ord} \\
\text{max}_0 \circ \text{OZ} & = \circ \\
\text{max}_0 \circ \text{OZ} & = \circ \\
\text{max}_0 (\text{O} \uparrow o_1) (\text{O} \uparrow o_2) & = \text{O} \uparrow (\text{max}_0 o_1 o_2) \\
\text{max}_0 (\text{OLim} c f) o & = \text{OLim} c (\lambda k. \text{max}_0 (f k) o) \\
\text{max}_0 o (\text{OLim} c f) & = \text{OLim} c (\lambda k. \text{max}_0 o (f k))
\end{align*}
\]

Long but straightforward proofs show that \( \text{max}_0 \) is monotone and computes and upper bound of its inputs. It reduces when given \( \text{O} \uparrow \) for both inputs, so it is strictly monotone. However, we cannot prove that it is a least upper-bound. The problem is that limits are not well-behaved with respect to the maximum. We could instead construct the maximum using \( \text{OLim} \), but this version would not be strictly monotone.

6.5.3.2 A Least Upper Bound

We solve the problems with \( \text{max}_0 \) using a type of sizes, which include only the subset of ordinals that are idempotent with respect to the maximum. We can then
define a type of sizes with the same interface as ordinals.

\[
\text{Size} : \text{Type} \\
\text{Size} = (o : \text{Ord}) \times (\max_0 o \ o \ o \leq o)
\]

\[
\sqrt{\cdot} : \text{Size} \to \text{Size} \to \text{Size} \\
s_1 \sqrt{s_2} = (\max_0 (\text{fst } s_1) \ (\text{fst } s_2), \ldots)
\]

\[
\text{SZ} : \text{Size} \\
\text{SZ} = (\text{OZ}, \leq_0 \text{Z})
\]

\[
\text{S} \uparrow : \text{Size} \to \text{Size} \\
\text{S} \uparrow s = (\text{O} \uparrow (\text{fst } s), \leq_\text{sucMono } (\text{snd } s))
\]

Critically, the sizes are closed under the maximum operation: if \(\max_0 o_1 \ o_1 \leq_0 o_1\) and \(\max_0 o_2 \ o_2 \leq_0 o_2\), then \(\max_0 (\max_0 o_1 \ o_2) \ (\max_0 o_1 \ o_2) \leq (\max_0 o_1 \ o_2)\). Zero and a successor operation for sizes are easily implemented. The difficulty is constructing a limit operator for sizes, since the self-idempotent ordinals are not closed under \(\text{OLim}\). Our trick is to take the limit of maxing an ordinal with itself. We assume we have a code \(\text{CN}\) whose elements have an injection \(\text{CtoN}\) into \(\mathbb{N}\). The natural numbers can be defined as an inductive type, but in our Agda development we add it as an extra code constructor. Having numbers lets us take the maximum of an ordinal with itself infinitely many times, resulting in an ordinal.
that is as large as the original but idempotent with respect to \( \text{max}_o \).

\[
\begin{align*}
n_{\text{max}} & : \text{Ord} \to \mathbb{N} \to \text{Ord} \\
n_{\text{max}} \circ Z &= 0Z \\
n_{\text{max}} \circ (S\ n) &= \text{omax} \ (n_{\text{max}} \circ n) \circ o \\
n_{\text{max}} & : \text{Ord} \to \text{Ord} \\
n_{\text{max}} \circ = \text{OLim} \ C\mathbb{N} \ (\lambda k. n_{\text{max}} \circ (\text{CtoN} \ k)) \\
n_{\text{max}} & : \{o : \text{Ord}\} \to \text{max}_o \ (n_{\text{max}} \circ o) \ (n_{\text{max}} \circ o) \leq_o (n_{\text{max}} \circ o) \\
\text{SLim} & : (c : \mathbb{C} \ell) \to (\text{El\Approx} \ c \to \text{Size}) \to \text{Size} \\
\text{SLim} \ c \ f &= (n_{\text{max}} \ (\text{OLim} \ c \ (\lambda k. \text{fst} \ (f \ k))), \ n_{\text{max}} \circ \text{idem})
\end{align*}
\]

Sizes satisfy all the same inequalities as raw ordinals, listed in Fig. 6.1. The monotonicity of \( \land \) follows from the monotonicity of \( \text{max}_o \), and the idempotence of \( \land \) follows by the definition of \( \text{Size} \). Monotonicity, idempotence, and transitivity of \( \leq_o \) together imply that \( \land \) is a least upper bound, and strict monotonicity follows from the strict monotonicity of \( \text{max}_o \).

### 6.5.4 Defining the Operations

With our type of sizes, we can outline our approach for proving termination for the approximate versions of our operations.

#### 6.5.4.1 Termination Strategy

We define a function \( \text{codeSize} : \mathbb{C} \ell \to \text{Size} \) which computes a \( \text{Size} \) in a straightforward way. Each term’s size is \( S\uparrow \) applied to the maximum of its subterms’ sizes. When a term binds a variable, we use \( \text{SLim} \) to take the limit over all values of that variable. Leaves in the syntax tree have size one, i.e. \( S\uparrow \text{SZ} \).
\[ \leq_\text{trans} : (s_1 : \text{Size}) \rightarrow (s_2 : \text{Size}) \rightarrow (s_3 : \text{Size}) \rightarrow (s_1 \leq_\text{s} s_2) \rightarrow (s_2 \leq_\text{s} s_3) \rightarrow (s_1 \leq_\text{s} s_3) \]

\[ \leq_\text{Z} : (s : \text{Size}) \rightarrow \text{SZ} \leq_\text{s} s \]

\[ \leq_\text{sucMono} : (s_1 : \text{Size}) \rightarrow (s_2 : \text{Size}) \rightarrow s_1 \leq_\text{s} s_2 \rightarrow S^\uparrow s_1 \leq_\text{s} S^\uparrow s_2 \]

\[ \leq_\text{cocone} : (c : \mathbb{C} \ell) \rightarrow (s : \text{Size}) \rightarrow (f : \text{ElApprox} c \rightarrow \text{Size}) \rightarrow (k : \text{ElApprox} c) \rightarrow s \leq_\text{s} f k \rightarrow s \leq_\text{s} \text{SLim} c f \]

\[ \leq_\text{limiting} : (s : \text{Size}) \rightarrow (c : \mathbb{C} \ell) \rightarrow (f : \text{ElApprox} c \rightarrow \text{Size}) \rightarrow ((k : \text{ElApprox} c) \rightarrow f k \leq_\text{s} s) \rightarrow \text{SLim} c f \leq_\text{s} s \]

\[ \vee \leq : (s_1 : \text{Size}) \rightarrow (s_2 : \text{Size}) \rightarrow (s_1 \leq_\text{s} s_1 \vee s_2) \times (s_2 \leq_\text{s} s_1 \vee s_2) \]

\[ \vee \text{mono} : (s_1 : \text{size}) \rightarrow (s_2 : \text{Size}) \rightarrow (s'_1 : \text{Size}) \rightarrow (s'_2 : \text{Size}) \rightarrow (s_1 \leq_\text{s} s'_1) \rightarrow (s_2 \leq_\text{s} s'_2) \rightarrow (s_1 \vee s_2) \leq_\text{s} (s'_1 \vee s'_2) \]

\[ \vee \text{idem} : (s : \text{Size}) \rightarrow (s \vee s) \leq_\text{s} s \]

\[ \vee \text{lub} : (s_1 : \text{size}) \rightarrow (s_2 : \text{size}) \rightarrow (s : \text{Size}) \rightarrow (s_1 \leq_\text{s} s) \rightarrow (s_2 \leq_\text{s} s) \rightarrow (s_1 \vee s_2) \leq_\text{s} s \]

**Figure 6.1:** Ordering on Sizes
To construct our operations, we use a record type:

\[
\text{record CastComp}(\ell : \mathbb{N}) (\text{notInUnkNeg} : \mathbb{E}) (\text{cSize} : \text{Size}) : \text{Type} \text{ where }
\]

\[
?\text{For}_{\text{æ}} : (c : C \ell) \to (\text{codeSize } c \equiv_{\text{Size}} \text{cSize}) \to \text{El}_{\text{æ}} c
\]

\[
\text{meet}_{\text{æ}} : (c : C \ell) \to \text{El}_{\text{æ}} c \to \text{El}_{\text{æ}} c \to \text{codeSize } c \equiv_{\text{Size}} \text{cSize} \to \ell_{\text{æ}} (\text{El}_{\text{æ}} c)
\]

\[
\text{codeMeet}^{\dagger} : (c_1 : C \ell) \to (c_2 : C \ell) \to (\text{codeSize } c_1 \lor \text{codeSize } c_2 \equiv_{\text{Size}} \text{cSize})
\]

\[
\to (c : C \ell) \times (\text{codeSize } c \leq_{\text{Size}} \text{codeSize } c_1 \lor \text{codeSize } c_2)
\]

\[
\text{cast}^{\dagger}_{\text{æ}} : (c_{\text{Dest}} : C \ell) \to (c_{\text{Source}} : C \ell) \to (\text{El}_{\text{æ}} c_{\text{Source}})
\]

\[
\to (\text{codeSize } c_{\text{Dest}} \lor \text{codeSize } c_{\text{Source}} \equiv_{\text{Size}} \text{cSize}) \to \ell_{\text{æ}} (\text{El}_{\text{æ}} c_{\text{Dest}})
\]

The boolean flag indicates whether we are currently traversing a subterm of a term of type \(?\). We explain the need for this in Section 6.5.5. The record’s fields are the operations we are building, with an additional proof that the size parameter is either the size of the code argument or the maximum of the two code sizes. So for any given code or codes, there is exactly one value of \text{cSize} where CastComp contains an operation that can be used.

To construct an instance of CastComp we use well-founded recursion. We show that there are no infinite descending chains of sizes using the usual accessibility predicate technique [18], then do well-founded recursion on the lexicographic ordering of \(\ell, \text{notInUnkNeg}\) and \text{cSize}. Thus, when writing the function for each operation, we can access a record containing the operations for every code size smaller than the current code, and for all smaller universes. We omit the proofs that each recursive call is on a smaller code, as well as the proofs about the size of composing codes: once we have determined the decreasing measure, the actual proofs are long but straightforward.

6.5.4.2 Composition

We describe the computational content of the meet function in Fig. 6.2. The meet for dependent pairs (and hence, inductive types) relies on casts. When we change the value of the pair’s first element, the type of the second element may change. Also, the call to codeMeet is not on a strictly smaller code, but on a strictly smaller
universe level.

In general, the non-trivial rules for composition are between terms built with the same syntax former whose subterms are metavariables. So the redex corresponds directly to the pattern we match upon in our model function. In the case of data constructors $D^C_1$ and $D^C_2$ for an inductive $C$, we perform an additional check that the constructors are the same, which is possible because propositional equality is decidable for finite numbers. Likewise, when composing two elements of $?\text{Type}_\ell$, we compare their heads using a similar decidable equality procedure. The reduct then contains compositions or casts of terms of smaller types. So in general, for a reduction $t_1 \& T t_2 \sim t$, $t_1$ and $t_2$ give the patterns for a clause of our function, and $E[t]$ gives the right-hand side. Since casts and compositions of subterms produce a result in the $L^\approx$ monad, the monadic bind can be used to incorporate them into the final result.

When we take the meet of two terms of type $?\text{Ty}$, we are no longer decreasing in the size of the codes, since the code is $C^?$ at each stage. However, because we have formulated $?\text{Ty}$ as a self-contained recursive type, we can use structural recursion to compose its values. Even though the domains of functions may not be of type $?\text{Ty}$, we never actually need to compose values of the domains, only of the return types. We omit this function, since each case is just a simpler version of a non-$?$ case.

The idea is largely the same for $\text{codeMeet}$ (Fig. 6.3), with the main difference being that the final result is not under a monad. However, recall that the function from $L$ can extract pure values from $L^\approx\text{Approx}$, because the monad was the identity. So whenever we use $\text{cast}_s$ or $\text{meet}_s$ when composing codes, we first approximate their inputs, ensuring that only pure computations are used to compute the codes.

### 6.5.5 Casting

#### 6.5.5.1 Same-Headed Codes

Figures 6.4 and 6.5 gives our implementation of casting. Similar to composition, the rules for casting generally are between types with the same head, where the subterms of the type are metavariables. Translating the reduct informs us as to how that case should be implemented. The interesting cases are for function types
meetₜ C? (?Tag h₁ x) (? Tag h₂ y) =
case (decEqₜ h₁ h₂) of
  (yes refl) ⇒ germmeetₜ x y
  (no npf) ⇒ pure ?U
meetₜ C U x y = pure unit
meetₜ C 0 x y = pure unit
meetₜ C ⊥ x y = pure (and x y)
meetₜ CType x y = pure (codeMeetₜ x y)
meetₜ (CΠ dom cod)f g = pure λx. let
  fx = fst (f x)
  gx = fst (f x)
  fgx = meetₜ(Approx (cod (approx x))) (approx fx) (approx gx)
  fgExact = do
    fEx ← snd(f x)
    gEx ← snd(g x)
    meetₜ(f (approx x)) fEx gEx
  in (fgx,fgExact)
meetₜ (CΣ dom cod) (x₁, y₁) (x₂, y₂) = do
  x₁₂ ← meetₜ x₁ x₂
  y₁' ← castₜ (cod (approx x₁₂)) (cod x₁) y₁
  y₂' ← castₜ (cod (approx x₁₂)) (cod x₂) y₂
  y₁₂ ← meetₜ (cod (approx x₁₂)) y₁' y₂'
  pure (x₁₂, y₁₂)
meetₜ C = c x y w₁ w₂ = meetₜ(Approx) w₁ w₂
meetₜ c x y = pure (.undefined)
  |--catch-all for remaining cases, produces error

Figure 6.2: Model of Composition
codeMeet\(^s\) (C?) \(c = c\)
codeMeet\(^s\) c (C?) = c
codeMeet\(^s\) (C∪) \(c = C\cup\)
codeMeet\(^s\) c (C∪) = C∪
codeMeet\(^s\) C₁ C₁ = C₁
codeMeet\(^s\) C₀ C₀ = C₁
codeMeet\(^s\) CType CType = CType

codeMeet\(^s\) (CΠ \(\text{dom}_1 \text{cod}_1\)) (CΠ \(\text{dom}_2 \text{cod}_2\)) = let
\(\text{dom}_{12} = \text{meet}^\text{\text{\~}}_{\text{\text{\~}}} \text{dom}_1 \text{dom}_2\)
\(\text{cod}_{12} = \lambda x_{12}. \text{let}\)
\(x_1 \leftarrow \text{cast}^\text{\text{\~}}_{\text{\text{\~}}} \text{dom}_1 \text{dom}_{12} x_{12}\)
\(x_2 \leftarrow \text{cast}^\text{\text{\~}}_{\text{\text{\~}}} \text{dom}_2 \text{dom}_{12} x_{12}\)
in codeMeet\(^s\) (\text{cod}_1 x_1) (\text{cod}_2 x_2)
in CΠ \(\text{dom}_{12} \text{cod}_{12}\)
codeMeet\(^s\) (CΣ \(\text{dom}_1 \text{cod}_1\)) (CΣ \(\text{dom}_2 \text{cod}_2\)) = \(\text{same as for CΠ}\)
codeMeet\(^s\) (C = c₁ x₁ y₁) (C = c₂ x₂ y₂) = let
\(c_{12} = \text{codeMeet}^s \text{c}_1 \text{c}_2\)
\(x'_1 = \text{cast}^\text{\text{\~}}_{\text{\text{\~}}} \text{c}_1 \text{c}_2 \text{c}_1 x_1\)
\(x'_2 = \text{cast}^\text{\text{\~}}_{\text{\text{\~}}} \text{c}_1 \text{c}_2 \text{c}_2 x_1\)
\(y'_1 = \text{cast}^\text{\text{\~}}_{\text{\text{\~}}} \text{c}_1 \text{c}_2 \text{c}_1 y_1\)
\(y'_2 = \text{cast}^\text{\text{\~}}_{\text{\text{\~}}} \text{c}_1 \text{c}_2 \text{c}_2 y_1\)
\(x_{12} = \text{fromL} \text{\text{\~}} (\text{meet}^\text{\text{\~}}_{\text{\text{\~}}} x'_1 x'_2)\)
\(y_{12} = \text{fromL} \text{\text{\~}} (\text{meet}^\text{\text{\~}}_{\text{\text{\~}}} y'_1 y'_2)\)
in C = c₁₂ x₁₂ y₁₂
codeMeet\(^s\) c₁ c₂ = C∪ --catch-all for remaining cases, produces error

Figure 6.3: Model of Composition for Codes
and equality types. For functions, we must cast the argument back to the source’s domain type, then cast the result. For equality, we cast the witness to the new type, but additionally take its meet with both of the new equated terms, to ensure that it is as precise as they are.

### 6.5.5.2 Functions and The Unknown Type

The only interesting cases for casting to or from \(?\) are for functions and inductive types, so we omit the helper functions \(\text{toGerm} \) and \(\text{fromGerm} \). In Fig. 6.6 we show the cases for casting from a function type to the germ and vice versa, which are the main places where the approximate and exact semantics differ.

To convert a function to the germ for function types, we need to produce an approximation of type \(?Ty\), and a function of type \(\triangleright_{\text{a}} \: ?Ty \rightarrow L_{\text{a}} \: ?Ty\). To compute the approximation, we apply the approximation in the input to \(?\text{For}\) of the domain type. This call is why \(?\text{For}\) is in our group of mutually dependent operations. To compute the non-approximated function, we branch based on whether we are in approximate or exact mode. In the approximate case the “full” function is just a duplication of the approximation, because \(\triangleright_{\text{a}} \: ?Ty = 1\). In the exact case, we can apply \((\text{pure next})\) to the guarded argument with \(\oplus\) to get a value of type \(\triangleright (L \: ?Ty)\), then apply \(\text{Later}\) to get \(L \: ?Ty\). The monadic bind then lets us give it as an argument to the exact function.

The key to logical consistency of the model is hidden in how we apply functions that take guarded arguments. Because \(?Ty\) \(\ell\) is defined as a guarded fixed-point, when applying a function taking a guarded argument, we must manually transport the argument in or out of the guarded type using \(\text{EqI}\). This accounts for how, when defining \(?Ty\), it could only refer to itself by a value of type \(\triangleright \text{Type}\), and hence had to apply \(\triangleright\) to obtain a type. This transport does not reduce, so the term itself does not run forever, even if it describes a computation that does.

When casting from \(?\), we check if the tag matches the type we are casting to. If it does, we use a helper function to cast to the destination type, and if it does not, then we produce an error. Converting from \(?\) to a function type is much simpler than the converse. We cast the argument to \(?Ty\), use \(\text{next}\) to guard the result of casting, then apply the germ function and cast the result. The result has the same
approximation for every value, regardless of the argument \( x \). This case is where approximation is lossy: a function can have an approximation for every input, but once we cast to \( ? \) and back, the result has a single approximation for all inputs.

### 6.5.5.3 Casting Between Inductives

When casting between inductive types and \( ? \) in either direction, we must handle the fact that the germ’s encoding of fields with function types may have arguments that are not of type \( ? \). As casts, these work just like casts between function types. The difference is in the termination argument, and is where the \texttt{notInUnkNeg} flag comes into play. When casting the results of the function application, we are casting between a code’s codomain and \( ? \text{Ty} \), giving a pair of codes strictly smaller than the function type and \( ?C \). However, no such guarantee exists for the arguments, because the germ’s arguments are not of type \( ? \text{Ty} \), and hence might have larger codes. We can take advantage of two facts. First, \( ?C \) should not occur anywhere within the code of the germ’s domain type, because of strict
positivity. Second, the germ’s domain type should be no more precise than the input code’s domain type, because the germ is supposed to be the least precise instantiation of a particular inductive type.

These two facts let us make a recursive call that is smaller in terms of the notInUnkNeg flag. If the flag is false, then we know we have already traversed into the left-hand side of a function arrow in the germ, so we should not be seeing the code C?. Likewise, because the domain type is as precise as the germ, it should also not contain C?. Internalizing this proof within the type theory is difficult, so
toGerm$_{\alpha}$ (CΠ dom cod) $f = \text{let} \newline f? = \text{fst} (f (\Pi \text{dom})) \newline f\text{Approx} = \text{fromL} (\text{cast}^\alpha_C (\text{cod} (\Pi \text{dom})) f?) \newline f\text{Exact} = \lambda(x_\alpha : \alpha \Rightarrow \exists Ty). \text{case } \alpha \text{ of} \newline \text{Approx} \Rightarrow \text{pure } f\text{Approx} \newline \text{Exact} \Rightarrow \text{do} \newline \hspace{1em} x_{\text{Src}} \leftarrow \text{Later} ((\text{next Now})@(\text{transport Eq}_x x_\alpha)) \newline \hspace{1em} x_? \leftarrow \text{cast}^\exists \text{dom } C? x_{\text{Src}} \newline \hspace{1em} fx \leftarrow \text{snd}(f x?) \newline \hspace{1em} \text{cast } C? (\text{cod} (\text{approx } x_?)) \text{ fx} \newline \text{in pure } (\exists \text{Tag } \Pi (\Pi (\lambda x. f\text{Approx}) f\text{Exact})) \newline \newline \text{fromGerm } (\Pi \Pi \text{dom cod} (\Pi \Pi \text{fLim fExact}) = \text{pure } \lambda x. \text{let} \newline \hspace{1em} \text{retApprox} = \text{fromL} (\text{cast} (\text{cod} (\text{approx } x)) C? (\text{fLim } \text{unit})) \newline \hspace{1em} x_? = \text{cast } C? \text{ dom } x \newline \hspace{1em} \text{retExact} = \text{do} \newline \hspace{2em} \text{let } x_\alpha = \text{next } x_? \newline \hspace{2em} fx \leftarrow \text{fExact} (\text{transport } (\text{sym Eq}_x) x_\alpha) \newline \hspace{2em} \text{cast } (\text{cod} (\text{approx } x)) C? \text{ fx} \newline \text{in (retApprox, retExact)} \newline \newline \textbf{Figure 6.6:} Casting Functions To and From the Germ
in this case we approximate the argument to \(?\) as a dummy value. If the flag is \(\text{true}\), we can make a recursive call on a larger code, because our overall metric is smaller in the lexicographic order.

The final detail to highlight is how we use the arity parameter of inductive signatures to handle inductive types that contain \(\text{?Ty} \rightarrow \text{?Ty}\) within a field. Because the constructor descriptions for inductive germs were defined before \(\text{?Ty}\), they cannot refer to \(\text{?Ty}\). So any fields that contain \(\text{?Ty} \rightarrow \text{?Ty}\) must be encoded in the germ by casting to type \(\text{?Ty}\), which ensures that any functions are approximated or guarded for the approximate or exact modes respectively. The arity number is how we track this when casting between an inductive and \(\text{?Ty}\). Since we know functions in inductive germs take exactly \(n\) parameters as specified in their arity, we process the first \(n\) parameters of a field, then convert whatever is left to \(\text{?Ty}\), function type or not.

### 6.5.5.4 The Unknown Term

Computing the unknown term for a given type is the simplest of our four operations, and follows the reduction rules quite directly. The notable case is computing the unknown term for an equality type, where we compose the equated terms to obtain the least precise witness of their consistency. The call to the meet here is the reason we must define \(\text{?For}\) as part of the mutual recursion, as opposed to defining it beforehand like with \(\text{UFor}\).

### 6.5.5.5 Removing Sizes

The final step in modelling eliminations is to use the record we constructed to expose the actual operations without size constraints. Suppose the record we constructed above using well-founded recursion is:

\[
\text{castComp : (ℓ : \mathbb{N})} \rightarrow (\text{notInUnkNeg : \mathbb{B}}) \rightarrow (\text{cSize : Size}) \rightarrow \text{CastComp \ell notInUnkNeg cSize}
\]
?For^s_{ae} C? = ??
?For^s_{ae} C\bot = \text{unit}
?For^s_{ae} C\top = \text{unit}
?For^s_{ae} C\bot\top = \text{true}
?For^s_{ae} CType = C?
?For^s_{ae} (C\Pi \text{ dom cod}) = 
\lambda x. (?For^s_{Approx} (\text{ cod } (?For^s_{Approx} \text{ dom})), \text{pure } (?For^s_{ae} (\text{ cod } (?For^s_{Approx} \text{ dom}))))
?For^s_{ae} (C\Sigma \text{ dom cod}) = (?For^s_{ae} \text{ dom}, ?For^s_{ae} (\text{ cod } (?For^s_{Approx} \text{ dom})))
?For^s_{ae} (C= c x y) = \text{ fromL } (\text{ meet^s_{Approx} c x y})
?For^s_{ae} (C\mu \text{ tyctor c D x}) = W?

\textbf{Figure 6.7: The Unknown Term for Codes}

Then we can define our operations by instantiating the size parameter with the code size or code size maximum, and providing \texttt{refl} for the equality proof.

?For^s_{ae} : (c : C \ell) \rightarrow \text{ El}_{ae} c
?For^s_{ae} c = (\text{ castComp } \ell \text{ false } (\text{ codeSize } c) ).?For^s_{ae} c \text{ refl}
\text{ meet}_{ae} : (c : C \ell) \rightarrow \text{ El}_{ae} c \rightarrow \text{ El}_{ae} c \rightarrow L_{ae} (\text{ El}_{ae} c)
\text{ meet}_{ae} c x y = (\text{ castComp } \ell \text{ false } (\text{ codeSize } c)).\text{ meet}_{ae} c x y \text{ refl}
\text{ codeMeet : C \ell \rightarrow C \ell \rightarrow C \ell}
\text{ codeMeet c_1 c_2 = }
\text{ fst } ((\text{ castComp } \ell \text{ false } (\text{ codeSize } c_1 \lor \text{ codeSize } c_2)).\text{ codeMeet}^s c_1 c_2 \text{ refl})
\text{ cast}_{ae} : (c_{Dest} : C \ell) \rightarrow (c_{Source} : C \ell) \rightarrow (\text{ El}_{ae} c_{Source}) \rightarrow L_{ae} (\text{ El}_{ae} c_{Dest})
\text{ cast}_{ae} c_{Dest} c_{Source} x =
\text{ (castComp } \ell \text{ false } (\text{ codeSize } c_1 \lor \text{ codeSize } c_2)).\text{ cast}_{ae}^s c_{Dest} c_{Source} x \text{ refl}
6.6 Inductives in the Model

The last missing piece of our model is specifying how to interpret inductive types in our model. As before, we assume that there is an a priori set of declared >GEQ inductive types. We now describe what codes and code-related functions for inductives we parameterize the translation over.

Each inductive has a code describing its parameters and indices. The parameter code is in the level one above the level of the inductive type. Having the parameter at a higher level allows types like List to take a Type₀ parameter while still living in Type₀.

\[
\begin{align*}
\text{Params} &: \text{CName} \rightarrow C (\ell + 1) \\
\text{Indices} &: \text{CName} \rightarrow \text{El}_\text{Approx} \text{Params} \rightarrow C \ell
\end{align*}
\]

Each inductive declaration is translated into a C Desc in the straightforward way: a CArg node describes the data constructor tags, then each non-recursive or recursive field is described using CArg or CRec respectively. This yields:

\[
\begin{align*}
\text{descFor} &: (\text{tyctor} : \text{CName}) \rightarrow (x_{\text{pars}} : \text{El}_\text{Approx} (\text{Params tyctor})) \\
&\quad \rightarrow \text{El}_\text{Approx} (\text{Indices tyctor} x_{\text{pars}}) \rightarrow C \text{Desc} \ 1 (\text{indSig tyctor})
\end{align*}
\]

6.6.1 Inductive Constructors and Eliminators

We can use interpDesc, descFor, and the W-type constructor Wsup to make generic constructors for each inductive type, whose arguments are codes denoting the >GEQ types of each field. The traversal to do so is straightforward and uses known techniques[48], but it is very long. We use the meta-notation ctorForC to indicate its construction, to avoid obfuscating the conceptual content of this chapter.

Our model also needs eliminators for inductive types. Thankfully, these can be computed from their descriptions, using well known techniques [48]. The computational content of the eliminators is straightforward, and follows the computational intuition of pattern matching by inspecting a constructor tag, then exe-
cutting a particular branch based on that tag, substituting in the stored values. Our only additions are handling $?^A$ and $U^B$. Consider an elimination where the motive is modelled by

$$c_{\text{motive}} : W D C \rightarrow C \ell$$

That is, it takes an element of the gradual $W$-type given by an inductive type’s description, and produces a code describing the result type for each input. Then, when given $W?^A$, our eliminator produces:

$$\text{pure (\text{For} (c_{\text{motive}} W?^A))}$$

Likewise, when given $WU^B$, we produce

$$\text{pure (\text{For} (c_{\text{motive}} WU^B))}$$

We use the meta-notation $\text{elimFor}_C$ to indicate the elimination form, again because the construction is both straightforward and not of specific interest for gradual dependent types.

Note: At the time of submission, generic inductive constructors and eliminators have not been implemented in our Agda mechanization, but since the technique for doing so is not novel, adding them should be straightforward, albeit time consuming.

### 6.7 Bringing it All Together

#### 6.7.1 The Translation

Figure 6.8 gives the final translation from $\triangleright \mathbf{GEQ}$ to $\mathbf{TCTT}$ using our definitions from the previous chapters.

#### 6.7.2 Key Properties

With our model complete, we present the final lemmas and theorem.

First, reducible terms translate to propositionally equal terms in the model.

**Lemma 6.7.1** (Soundness). If $\Gamma \vdash t \iff T$ and $t \leadsto_{\mathcal{E}} t'$ then

$$\exists_{k : L^a_{\mathcal{E}}} \exists_{\mathcal{T}(x_k)} \rightarrow (\mathcal{E}_k[t]\mathcal{E}[t'])$$

is inhabited in $\mathbf{TCTT}$. Moreover, $\mathcal{E}_k[t]$ is
\[ T[\text{Type}_t] = \text{ CType} \]
\[ T[\text{?Type}_t] = \text{ C?} \]
\[ T[\text{UType}_t] = \text{ C U} \]
\[ T[(x : T_1) \to T_2] = \text{ CΠ} \, T[[T_1]] \left( \lambda x. T[[T_2]] \right) \]
\[ T[[C(t_{\text{params}}, t_{\text{indices}})]] = (\text{Cµ C (Indices C)} \right)
\begin{align*}
&\text{descFor (from } \mathcal{L} \, E[[t_{\text{params}}]) \text{ (from } \mathcal{L} \, E[[t_{\text{indices}}]))} \\
&\text{(from } \mathcal{L} \, E[[t_{\text{indices}}])))
\end{align*}
\]
\[ E[[\text{Type}_t]]_a = \text{ pure CType} \]
\[ E[[\text{(x : T}_1) \to T_2]]_a = \text{ pure (CΠ } T[[T_1]] \left( \lambda x. T[[T_2]] \right) \text{)} \]
\[ E[[C(t_{\text{params}}, t_{\text{indices}})]]_a = \text{ pure (Cµ C (Indices C)} \right)
\begin{align*}
&\text{descFor (from } \mathcal{L} \, E[[t_{\text{params}}]) \text{ (from } \mathcal{L} \, E[[t_{\text{indices}}]))} \\
&\text{(from } \mathcal{L} \, E[[t_{\text{indices}}])))
\end{align*}
\]
\[ E[[C(t_{\text{params}}, t_{\text{indices}}, t_{\text{args}})]]_a = \text{ ctorFor } \text{C (from } \mathcal{L} \, E[[t_{\text{params}}]) \text{ (from } \mathcal{L} \, E[[t_{\text{indices}}])) \text{ E[[t_{\text{args}}])} \\
\]
\[ E[[\text{ind}_C(t_{\text{scrut}}, T_{\text{P}}, t_{\text{rhs}})]]_a = \text{ elimFor } \text{C (from } \mathcal{L} \, E[[t_{\text{scrut}}]) \text{ E[[t_{\text{P}}]) \text{ E[[t_{\text{rhs}}])} \\
\]
\[ E[[\lambda (x : T). t]]_a = \text{ pure } \lambda (x : E_{\text{t}} [[T]]) \text{ (from } \mathcal{L} \, E[[t]]_{\text{approx}, E[[t]]_a}) \\
\]
\[ E[[t_1 \, t_2]]_a = \text{ bind}_L^L_{\text{a}} (\lambda f \, \lambda x. \text{ snd } (f \, x)) \, E[[t_1]] \, E[[t_2]] \\
\]
\[ E[[T_1 \, \langle T_2 \rangle]]_a = \text{ bind}_L^L (\text{cast}_{\text{a}} T[[T_2]] \, T[[T_1]]) \, E[[T]] \\
\]
\[ E[[h]_a T]]_a = \text{ bind}_L^L (\text{cast}_{\text{a}} \text{ C? } T[[\text{germ}(h)]]) \, E[[T]] \\
\]
\[ E[[T_1 \& T_2]]_a = \text{ bind}_L^L (\text{meet}_{\text{a}} T[[T]]) \, E[[T_1]] \, E[[T_2]] \\
\]
\[ E[[T \, t_1, t_2, t_{\text{P}}]]_a = \text{ do } T_1 \leftarrow \text{ snd } (\text{bind } E[[T]] \, E[[t_1]]) \\
T_2 \leftarrow \text{ snd } (\text{bind } E[[T]] \, E[[t_2]]) \\
\]
\[ E[[T \, t_{\text{P}}]]_a = \text{ cast } T_2 \, T_{\text{W}} \, t_{\text{P}} \]

**Figure 6.8:** Syntactic Model of $\ast$GEQ: Summary
reducible.

See proof in Appendix A.

Second, we note that the translation produces well typed terms.

**Lemma 6.7.2 (Model Preserves Types).** If $\Gamma \vdash t \iff T$ then

$$E[t] : (x_k : \mathcal{L}_{\approx} \mathcal{T} \Gamma(x_k)) \to (\mathcal{L}_{\approx} \mathcal{T} T) \text{ in } TCTT.$$  

See proof in Appendix A.

The isomorphism between $\mathcal{L}_{\text{Approx}} A$ and $A$, combined with type safety and the fact that each $\triangleright GEQ$ reduction corresponds to one or more reductions, gives us our strong termination result, from which decidable type checking follows.

**Theorem 6.7.3 (Strong Approximate Normalization).** If $\Gamma \vdash t \iff T$, then for some normal form $v$, $t \rightarrow^{*}_{\text{Approx}} v$.

**Corollary 6.7.4 (Decidable Checking).** Type checking in $\triangleright GEQ$ is decidable.

### 6.7.3 A Final Example

Since $\triangleright GEQ$ is the culmination of the ideas presented in this thesis, we discuss its limitations and possible future work in Chapter 7. Instead, we conclude the chapter showing how the $\triangleright GEQ$ handles $\Omega$, the quintessential non-terminating term. Define:

$$F := \lambda(x : ?_{\text{Type}}) . (\langle ?_{\text{Type}} \rightarrow ?_{\text{Type}} \iff ?_{\text{Type}} \rangle x) \ x$$
Then $\Omega := F F$. The exact translation of $F$ in our model is (after some let-bindings and inlinings for readability):

\[
F : ?Ty \ell \rightarrow ?Ty \ell \\
F = \lambda(x : ?Ty \ell). \text{let} \\
\quad fA = \text{from}\mathcal{L}(\text{cast}_\text{Approx} (\Sigma \Pi \Sigma? (\lambda\_?)) \Sigma? x) \\
\quad \text{retA} = \text{from}\mathcal{L}(\text{snd}(fA \ x)) \\
\quad \text{retE} = \text{do} \\
\quad \quad fE \leftarrow \text{cast}_\text{Exact} (\Sigma \Pi \Sigma? (\lambda\_?)) \Sigma? x \\
\quad \quad \text{snd}(fE \ x) \\
\text{in} (\text{retA}, \text{retE})
\]

To apply $F$ to itself, we must inject the argument into $?Ty \ell$, that is, using:

\[
F? := \text{cast} \Sigma? (\Sigma \Pi \Sigma? (\lambda\_?))
\]

Substituting in and expanding the definition of $\text{cast}$ yields:

\[
\beta^* \quad (\Sigma\text{Tag} (\Sigma\Pi) (\lambda\_?) (\lambda x_\Delta. \text{do} \\
\quad x_? \leftarrow \text{Later} ((\text{next} \text{Now})(\text{transport} \text{Eq} x_\Delta)) \\
\quad fE \leftarrow \text{cast}_\text{Exact} (\Sigma \Pi \Sigma? (\lambda\_?)) \Sigma? x_? \\
\quad \text{snd} (fE \ x_?) ) )
\]

To inject $F$ into $?Ty$, we compute an approximate and exact version. For the approximate version, $??$ replaces $x$ in $F$. In the function position, it is cast to $?For$ at the function type $?Ty \rightarrow ?Ty$. This is the constant function that produces $? as the result for all arguments, so the approximate part of $F?$ always produces $?$. For
the exact version, \( F \) is converted to a function that takes a guarded argument, so we use the monadic bind to apply \( F \) to the guarded argument.

Now we create \( \Omega \) by applying \( F \) to \( F? \). Again the approximate part yields a ?? on every input. Let us examine the reduction of the exact part:

\[
\text{snd} \ (F \ F?) : \mathcal{L} \ (?Ty \ ?)
\]
\[\longrightarrow^{*}(1)\]
\[\text{do}
\]
\[\ fE \leftarrow \text{cast}_{\text{Exact}} \ (\Pi \ C \ ? \ (\lambda \_ \ C \ ?)) \ C \ ? \ F?
\]
\[\text{snd} \ (fE \ F?)\]
\[\longrightarrow^{*}(2)\]
\[\text{do}
\]
\[\ fE \leftarrow \text{pure} \ \lambda x. (??, do
\]
\[\ x_? \leftarrow \text{Later} \ (((\text{next} \ \text{Now}) \circ (\text{transport} \ \text{Eq} \ (\text{transport} \ (\text{sym} \ \text{Eq}) \ (\text{next} \ x))))))\]
\[\ fE' \leftarrow \text{cast}_{\text{Exact}} \ (\Pi \ C \ ? \ (\lambda \_ \ C \ ?)) \ C \ ? \ (\text{transport} \ (\text{sym} \ \text{Eq}) \ (\text{next} \ x))\]
\[\text{snd} \ (fE' \ (\text{transport} \ (\text{sym} \ \text{Eq}) \ (\text{next} \ x))))\]
\[\text{snd} \ (fE \ F?)\]
\[\longrightarrow^{*}(3)\]
\[\text{do}
\]
\[\ x_? \leftarrow \text{Later} \ (((\text{next} \ \text{Now}) \circ (\text{transport} \ \text{Eq} \ (\text{transport} \ (\text{sym} \ \text{Eq}) \ (\text{next} \ F?))))))\]
\[\ fE' \leftarrow \text{cast}_{\text{Exact}} \ (\Pi \ C \ ? \ (\lambda \_ \ C \ ?)) \ C \ ? \ x_?\]
\[\text{snd} \ (fE' \ x_?)\]

At this point, reduction stops, because the monadic bind cannot reduce on the transported version of \( \text{next} \ F? \). We can, however, apply a fact about propositional equality: transporting a term by a proof, then back by \( \text{sym} \) of that proof, yields
the original term. This gives us:

\[
\begin{align*}
\text{do} \\
\quad & fE' \leftarrow \text{cast}_{\text{Exact}} \ (\text{C} \Pi \ (\lambda \_ \ C ?) \ C ? \ F ?) \\
\quad & \text{snd} \ (fE' \ F ?)
\end{align*}
\]

This is exactly the reduct of \( \rightarrow^* \) (1). So we can unfold the transport of Eq\(\ast\) an arbitrary number of times, and always end up with the same term, which is exactly how we expect \( U \) to behave.
Chapter 7

Discussion

To conclude this dissertation, we discuss the main issues, limitations, and possible future work surrounding our research.

7.1 Does Approximate Normalization Matter?

One question that has gone unanswered until this point is whether we should actually care about approximate normalization. Lennon-Bertrand et al. [87] present two languages satisfying the gradual guarantees without approximate normalization, one without decidable type checking and one with restrictions on the universes of types in dependent functions. Not all static dependently typed languages have decidable type checking. Proving approximate normalization required significant machinery, both in the Chapter 3 and Chapter 6 versions. And the end result is a programming language where some faulty programs could be rejected with static type errors, but are instead accepted.

We argue that approximate normalization is worthwhile. In a talk, Sterling [139] notes that, while the time complexity bound on dependent type checking is high, in practice it can be done quite quickly. Keeping type checking decidable could help ensure that gradual dependent types can be checked quickly for most real-world programs. Moreover, decidable type checking lets gradual dependent types integrate more easily into the existing ecosystem for dependently typed programming. In Chapter 6, we mentioned how decidable type checking allowed
gradual dependent types to compile to existing core languages without needing to modify them. We suspect that this benefit will extend when integrating tools like proofs search or SMT into a gradual dependently typed language.

Even if future gradual dependently typed languages do not use approximate normalization, the knowledge gained from developing it will be useful to the future of gradual dependent types. Gradual types introduce effects into dependent types’ normally pure conversion checking, and some technique must be used to handle them. For instance, our model could be adapted using exact but undecidable conversion checking if, for instance, we introduced an operator with the type \( L^\alpha \text{Type} \rightarrow \text{Type} \). But the framework of dividing the model into effectful and non-effectful portions would still be useful with a different model of type checking.

By exposing the compromises necessary for decidable checking with gradual dependent types, we hope to give the programming languages community the information necessary to determine whether the benefits of decidable checking justify the compromises.

7.1.1 Approximate Witnesses

One design decision that deserves further exploration is whether approximate normalization should be used to track witnesses of consistency. In GDTL, all type comparisons happen with normal forms, so evidence must use approximate semantics, since exact execution does not normalize under binders. \( \rightarrow \text{GEq} \) inherited this design choice. However, since its dynamic and static semantics differ only in casts to and from \(?\), it is not strictly necessary to use approximate witnesses.

With approximate witnesses, the question is whether witnesses are too opaque to the programmer to allow non-termination. The danger is that the programmer could write \(?\) in the surface language and have it elaborate into a non-terminating term. For example, if the programmer writes \(?\) in a context expecting a proof of \( y (\lambda x. x x) \equiv y (\lambda x. x x) \), this elaborates with the witness \( y (\lambda x. x x) (\lambda x. x x) \). If \( y \) is ever instantiated to \( \lambda f. f x \), the witness will run forever. The programmer would be debugging an infinite loop where the source of the bug is the types, not the term.
However, the above example is highly contrived, and it is possible that all such examples are. The use of approximate witnesses is not without downsides. It means that \( \triangleleft \text{GEq} \) does not satisfy the EP-pairs property, which was useful for proving that casts did not needlessly produce \( ? \) as a result. Moreover, it means that there are programs that either never raise a dynamic error when a static equality is violated, or they raise it later than is necessary, because information is lost to approximation. The issue is one of ergonomics, and will likely not be resolved until there is an implementation of a gradual programming language with at least a few users.

### 7.2 Limitations and Future Work

We present our limitations and future work simultaneously, because each limitation of our contributions comes with a suggestion for future work to address the limitation.

#### 7.2.1 Adequacy

Though we established the soundness of our model in Chapter 6, we did not prove another important property of denotational models: adequacy. A model is adequate if the translations of two terms are equal in the model if and only if they are observationally equivalent in the source language. Adequacy establishes that there are not too many equalities in the target. For example, if we were to map every source type to the same target type, and every value to the same target value, our model would be sound but not adequate. When a model is adequate, theorems about the model can be easily translated back to the source language. We conjecture that our model is adequate, but proving it is left to future work.

#### 7.2.2 Approximating Inductive Types

In Chapter 6, we modelled the germ of inductive data types by casting every field to \( ? \) or a function producing \( ? \). Doing so made the termination argument easy, but because witnesses are stored in approximate form, converting terms to \( ? \) needlessly forgets consistency information in equality proofs. We hope that in the fu-
ture, Agda’s termination checker can be refined to allow for \( ?Ty \) to refer to types other than itself in function codomains. More study of induction-recursion and its implementation in Agda should help find a solution to this issue.

In general, the current handling of inductive types is unsatisfactory, since it assumes either a programmer or mechanical system will generate the relevant instances for each inductive type. The problem is that in \( \mathsf{GCIC} \) and its derivatives, the germ of an inductive type relies on replacing its parameters with \( ? \). The presentations of generic inductive types we used do not treat parameters at all: a parameterized type is just a function producing a description or container. Further exploration into generic ways of representing inductives could let our model use a completely closed type theory.

### 7.2.3 Mechanized Metatheory

Because Chapter 6 provided a model of both approximate and exact normalization, it has potential far beyond proving strong normalization. Having a semantic model of \( \triangleright \mathsf{GEq} \) could let future developments move past complex simulation-based arguments for the gradual guarantees. We conjecture that a proof technique similar to the EP-pairs used by Lennon-Bertrand et al. [87], New and Ahmed [105] can be adapted to \( \triangleright \mathsf{GEq} \), but using Galois connections instead of EP-pairs. Because \( \triangleright \mathsf{GEq} \) uses approximate normal forms for equality witnesses, it violates the full EP-pairs property: casting to \( ? \) then back can reduce the precision of a witness. To show that precision in terms is not needlessly lost, we could establish a Galois connection for a more restrictive consistency relation, where terms are compared purely based on their error behaviour, and \( ? \) is not the top of the precision lattice. This would establish that reducing an exact term’s precision may cause it to raise an error in fewer contexts, but if the term terminates without error, the final value is unchanged. Likewise, semantic techniques could be used to show that composition computes a true meet for precision, guaranteeing that precision is never needlessly lost when composing terms.

A major limitation of Chapter 5 is that, despite designing the language around preserving static equivalences, we never actually establish that we preserve static equivalences. The most general version of this property is full abstraction, which
says that all observationally-equivalent static terms are gradually equivalent. However, the usual technique to prove full abstraction does not scale well to gradual dependent types. The problem is this technique requires simulating the gradual language in the static language in a way that produces the same effects, but static dependent types lack effects. However, the semantic model of Chapter 6 could be used to prove a weaker version of the theorem, which is that all propositionally equal static terms are observationally equivalent in the gradual language. If static terms are propositionally equal, then the equality proof could be lifted into the gradual language as a fully-precise witness. If we proved that composition computed a greatest lower bound, then we would know that the composing two static terms produced an equi-precise term. If we proved that all equi-precise terms were equal in the model, this would establish that the two terms were equal.

7.2.3.1 Simplifying the Model

Our current model of gradual dependent types involves one giant mutually-recursive type of codes, with the elimination operations defined mutually with each other. This technique is complex, and frankly, has resulted in a model that works, but is difficult to read and understand. Directly modelling $\llt{\text{GEq}}$ in GTT was useful for proving termination, but for the broader goal of reasoning about the metatheoretic properties of $\llt{\text{GEq}}$, we could use the more conventional model-theoretic approach of defining a semantic domain denoting the types of $\llt{\text{GEq}}$, then showing that it is closed under semantic versions of $\llt{\text{GEq}}$’s type formers. Such an approach should either help reduce the mutual dependencies, or at least reveal where and why they are strictly necessary.

7.2.4 Symbolic Gradual Dependent Types

Our gradual dependently typed languages have a number of limitations that can all be addressed with symbolic means. We believe that there is a deep connection between gradual dependent types and symbolic execution waiting to be explored. We outline the potential connections below.
7.2.4.1 Remembering Constraints

In Chapter 5, we touted the use of equality witnesses for catching bugs by dynamically tracking equalities. Unfortunately, the version of equality there does not capture all equality constraints that would be implied in a static program. As we discussed in that chapter, the problem is that witnesses remember constraints across time, not space. That is, if an equality proof is copied, there is no memory that the two copies referred to the same value. Likewise, if an equality proof is transformed by applying a function to it, and constraints are learned about the result of that function, then no constraints are learned about the original proof.

We propose that monotonic references [137] could be integrated with gradual propositional equality, so that constraints were remembered across both time and space. For non-dependent gradual types, monotonic references were developed for statefully tracking precision information that increases as a program runs. Because arbitrary terms can be equated, a system of dynamic higher-order unification would need to be developed to track constraints between gradual terms.

7.2.4.2 Pattern Matching and Gradual Unions

Our notion of pattern matching on ? is needlessly lossy. In Chapter 3, we justify ? t reducing to ? in terms of AGT: because every possible static function is denoted by ?, every result is possible. When eliminating ? as a member of an inductive type, we also produce ? as a result. However, the AGT justification does not apply in this case: for a given elimination, the possible outcomes are restricted by the branches given to the eliminator. For example, the term if x then 0 else 1 never produces 2 for any x. So why should if ? then 0 else 1 denote every possible term?

We propose that this issue could be solved with a dependent version of gradual union types [24, 145] and gradual recursive types [133]. The set of possible outcomes of an elimination could be modelled as the recursive union of all branches of the eliminator, where each recursive invocation of the eliminator is replaced with a recursive self-reference.

As is expected with gradual dependent types, some compromise would be necessary. The challenge is in comparing recursive unions of gradual terms. Determining if two recursive unions of terms are consistent is likely to be undecid-
able, just as it is undecidable to determine if two functions agree on every input. However, set constraints [5, 115] or regular tree languages [25, 36] could be used to provide an upper-bound on the possible results of an elimination.

7.2.4.3 Gradual AGT

Unfortunately, Chapters 5 and 6 do not provide principled justification for their interpretation of gradual types. In Chapter 3, we provide such justification using AGT, but with a very limited calculus.

We propose that a syntactic model in the style of \texttt{GEq} could be used to define an AGT-style gradual language with inductive types and propositional equality. For \texttt{GEq}, we use a type to model the values of \texttt{GEq}, then use monadic functions to model computations on terms and casts between types. A similar approach could be used to model AGT-style evidence based semantics instead of casts. A monad-like type $M : C \ell \rightarrow \text{Type}$ could track gradual computations, where a term of type $M c$ contained a code $c'$, an inhabitant of $\text{El } c'$, and evidence that $c$ and $c'$ were dynamically consistent.

7.2.4.4 Gradual Symbolic Execution

Monotonic references, gradual unions, and AGT suggest a deeper connection between gradual dependent types and symbolic execution. Under the AGT interpretation, executing an imprecise program could be interpreted as determining what values are in the set of results of executions of the static programs denoted by the gradual program’s concretization. Symbolic execution is a technique to analyze what the set of results is for a set of programs, where the input set is represented as a single symbolic static program. So a gradual dependently typed program corresponds to a symbolic static dependently typed program, where the imprecise parts of the gradual program correspond to the symbolic parts of the static program. We suspect that techniques like choice calculi [31, 62] and higher-order symbolic execution [106] could aid in executing gradual programs, more precisely handling branching and pattern matching.
7.2.5 Broader Applications of Approximate Normalization

In Chapters 3 and 6, we used approximate normalization as a way to handle non-termination as an effect in terms on which types depend. This technique could provide a general way to handle effectful code in dependent types: if a type depends on an effectful term, that term can be approximated to ? and checked as a gradual term, with any deferred checks inserted at run-time. For example, one could have a proof that \texttt{readInt > 0}, which was treated at compile time as a proof that ? > 0, but at run time read an integer from stdin and ensured that its value was greater than zero.

7.2.6 Implementation

The translation of Chapter 6 lends itself well to taking a gradual surface language and translating to the core language of an existing implementation of dependent types. For example, as of 2023, the Idris 2 developers are in the midst of developing a new core language that would be an ideal target for such translation. Similarly, systems like macros in Lean [102] or Cur [28], or Agda’s tactic arguments, could support a lightweight gradual language.

Several issues would need to be addressed for such an implementation to be usable. Dependent types are extremely verbose without implicit arguments, but implicits are resolved with unification. So a full implementation would need a unifier that is aware of consistency when solving for metavariables, possibly adding higher-order unification to the work of Siek and Vachharajani [134] Garcia and Cimini [66].

Likewise, languages like Idris rely heavily on typeclasses. In general, gradual handling of typeclasses is an open problem, even without dependent types. However, the ability to model imprecision in both terms and types could help with finding a solution. For example, typeclasses could integrate with the gradual unions of terms above. Each typeclass dictionary could take an extra parameter, equating the actual typeclass parameters to those specified for the particular implementation. When calling a typeclass method on an imprecisely typed term, one could supply the gradual union of all available implementations of the class. Then, as type information was gained, the available implementations would nar-
row as the extra equality parameters reduced to errors for some implementations.

7.3 Conclusion

When we began this track of research, all major developments in gradual dependently typed languages were focused around subset types and boolean predicates. Our work has shown that a full spectrum gradual dependently typed language is feasible, where \( \_ \) enjoys first-class status as a term. Allowing \( \_ \) to be used as a term results in a highly granular notion of precision for indexed types. We have only begun to explore how the hole-based programming of dependent types can mesh with REPL and test-based development, but we now have the theoretical foundation to make that exploration possible.

That said, the major lesson of this dissertation is one of compromise, and we suspect this will continue to be at the heart of the story of gradual dependent types. To achieve decidable type checking, we needed to compromise on what it meant for two types to be convertible. To achieve run-time consistency tracking, we needed to compromise what it meant for two terms to be dynamically consistent. Our work shows that these are concessions, not defeats.

Software is ubiquitous, and it seems that it will be for the foreseeable future. Release cycles for software tend to be short, and there is pressure to write and release software quickly. Furthermore, it seems that tools to automatically generate code are here, whether or not we have the tools to specify how that code should behave and to verify whether it meets that specification. Dependent types provide a unified, principled, theoretically justified framework for specification and verification, and gradual dependent types extend that framework to dynamic checking. We have not removed the barrier to entry for dependent types, but we have lowered it. Our hope is that this in turn helps reduce the barriers to writing safe, dependable code.
Bibliography


245


255


259


260


Appendix A

Full Proofs

**Theorem 3.4.1 (Soundness).** For all $\emptyset \not\subseteq S \subseteq SNF$, $S \subseteq \gamma(\alpha(S))$.

*Proof of Theorem 3.4.1.* We assign a depth number to each set $S \subseteq SNF$. Define:

- $D(S) = 0$ if $S$ contains two or types built with different head constructors.
- $D(\{C \ t_1 \ldots \ t_n \mid t_1 \in S_1, \ldots, t_n \in S_n\}) = \max_{S' \subseteq S_1 \ldots S_n} (D(S'))$ for a head constructor $C$.

We perform induction on this $D(S)$, proving the corresponding result mutually for $\alpha_N$ and $\gamma_N$. The cases are nearly identical, so we highlight the core technique. We also note that in the definitions, $\alpha_N$ and $\gamma_N$ only ever take argument sets where all the neutrals have the same head variable.

- **Case** $D(S) = 0$:
  
  Then $\alpha(S) = \emptyset$, so $\gamma(\alpha(S))$ is SNF $\supseteq S$.

- **Case** $D(S) = 1 + n$:
  
  Then all terms in $S$ are constructed with the same head. Proceed by cases on this head.

- **Case** $S = \{Type\}$:
Then \( \gamma(\alpha(\{\text{Type}_t\})) = \gamma(\text{Type}_t) = \{\text{Type}_t\} \supseteq \{\text{Type}_t\} \). Same reasoning holds for \( x \).

**Case \( S = \{\lambda x.t \mid t \in S'\} \):**

If \( S \) is nonempty then so is \( S' \). So \( \alpha(S) = \lambda x.\alpha(S') \). Our hypothesis gives that \( \gamma(\alpha(S')) \supseteq S' \), since \( S' \) has a depth bounded by \( n \). So \( \gamma(\alpha(S)) = \gamma(\lambda x.\alpha(S')) = \{\lambda x.t' \mid t' \in \gamma(\alpha(S'))\} \). Since \( \gamma(\alpha(S')) \supseteq S' \), our result holds. The reasoning is similar for the remaining cases.

**Theorem 3.4.2 (Optimality).** For \( \emptyset \not\subseteq S \subseteq \text{SNF} \) and \( U \in \text{GNF} \), if \( S \subseteq \gamma(U) \) then \( \alpha(S) \subseteq U \). That is, \( \gamma(\alpha(S)) \subseteq \gamma(U) \).

**Proof of Theorem 3.4.2.** By induction on \( r \), proving the corresponding result mutually for \( \alpha_N \) and \( \gamma_N \).

**Case \( \text{Type}_t \):**

Then \( \gamma(u) = \{\text{Type}_t\} \), so if \( S \) is nonempty then \( S = \{\text{Type}_t\} \), so \( \alpha(S) = \text{Type}_t \subseteq \text{Type}_t \). Same reasoning holds for \( x \).

**Case \( ? \):**

Trivial: \( \alpha(S) \subseteq ? \) since \( ? \) is the least precise term.

**Case \( \langle N' \rangle N \):**

Then \( \gamma(\langle N' \rangle N) = \{\langle N' \rangle N \mid N' \in \gamma_N(N'), N \in \gamma(N)\} \). \( \gamma_N \) produces terms with the same head for both \( N \) and \( N' \). If \( S \) is nonempty, then it must be of the form \( \{\langle N' \rangle N \mid N' \in S_1, sN \in S_2\} \). So \( \alpha(S) \) has the form \( \langle N' \rangle S \). By our induction hypothesis, similar reasoning holds for the remaining cases.

**Lemma A.0.1 (Composition preserves typing).** Suppose \( \Gamma \vdash N_1 \Rightarrow N_1 \) and \( \Gamma \vdash N_2 \Rightarrow U_2 \). If \( N_1 \cap N_2 = N \) and \( U_1 \cap U_2 = U \) then \( \Gamma \vdash N \Rightarrow U \).

Suppose \( \Gamma \vdash u_1 \Leftarrow U \) and \( \Gamma \vdash u_2 \Leftarrow U \). If \( u_1 \cap u_2 = u \) then \( \Gamma \vdash u \Leftarrow U \).
Proof of Lemma A.0.1. We do mutual induction on the two above statements.

For neutrals:

- **Case** Variable:
  
  Trivial because $x \cap x = x$

- **Case** Application:
  
  Straightforward induction shows that $\cap$ distributes over substitution, so the substituting the meet of the arguments produces the meet of the results of substitution. The result then follows from IH on the arguments.

For normals: straightforward induction. Most cases are straightforward from IH. The one interesting case is:

- **Case** $\langle \varepsilon_1 \rangle N_1 \cap \langle \varepsilon_2 \rangle N_2$:
  
  The resulting evidence is valid because composition produces a term that is as precise as both inputs. Likewise, by our premise $N_1$ and $N_2$ check against the same type $U$, so they must synthesize types $U_1$ and $U_2$ that are both consistent with $U$. By our premise the meet of the evidence $\varepsilon_1 \cap \varepsilon_2$ is defined, and since the meet is monotone, this combined evidence is as precise as, and hence consistent with, $U$. Again, since the meet is monotone, $U_1 \cap U_2$ must be defined and $\varepsilon_1 \cap \varepsilon_2 \subseteq U_1 \cap U_2$. So then $U_1 \cap U_2$ is consistent with $U$, giving us what we need to preserve the typing relation.

\[ \square \]

Lemma A.0.2. If $\Gamma_1 \vdash u_1 \Leftarrow U_1$ and $\Gamma_1 \vdash (U_2 \Leftarrow u_2) \Downarrow u_2$, then $\Gamma_2 \vdash u_2 \Leftarrow U_2$.

If $\Gamma_1 \vdash N_1 \Rightarrow U_1$ and $\Gamma_1 \vdash (N_2 : U_2) \Rightarrow (N_2 : U_2)$, then $\Gamma_2 \vdash N_2 \Rightarrow U_2$.

Proof of Lemma A.0.2. We prove the two statements mutually, by induction on the derivation of the casting or re-typing relation.

- **Case** Neutral retyping variables:
  
  Trivial from premise, since types are in $\Gamma$.

- **Case** Neutral retyping application:
The premises mirror the premises of the typing rule for neutral applications. By our IH, $u'$ produces a well-formed output because it is the result of casting $u$ to the type of the variable in $U'$. The final result type is computed using codomain substitution, which is exactly how the synthesis rule for spine application produces its result.

**Case Normals**:

Straightforward induction in each case: the casts in the premises were chosen precisely to preserve typing.

---

**Theorem 3.5.1** (Normalization preserves typing). If $\Gamma \vdash u \leftarrow t \Leftarrow U$, then $\Gamma \vdash u \Leftarrow U$.

*Proof of Theorem 3.5.1.* By induction on the normalization derivation, mutual with the following: if $\Gamma \vdash i \leadsto u \Rightarrow U$, then $\Gamma \vdash u \Leftarrow U$. Note that synthesis is not preserved by normalization, only checking.

Straightforward induction establishes that hereditary substitution preserves typing, i.e. replacing $x$ by $u$ in the input produces a result typeable by replacing $x$ with $u$ in the input type.

For normalization, most cases are trivial, because the premises of normalization are directly mirror the premises of normal typing. The exceptions are below:

**Case GNSYNTXDYN**:

The normalization rule is a synthesis rule, but the typing rule is a checking rule. However we can type $\_\_\_\_\_\_$ at any type with GWFUnk, which gives our result.

**Case GNSYNTXType**:

As above, normalization is a synthesis rule, but types are checked in normal forms, but because GNSYNTXType synthesizes the universe at the level that GWFTYPE expects, we can type the result.

**Case GNSYNTXAnn**:

266
We have by our premise that \( u \) is the result of normalizing at type \( U \) (i.e. the synthesized type), so IH gives that \( u \) checks against the synthesized type \( U \), so the result is immediate.

\[ \blacktriangleright \text{Case GNC/h.sc/e.sc/c.sc/k.scS/y.sc/n.sc/t.sc/h.sc} : \]

Our hypothesis gives that \( u \) checks against \( U \), which is consistent with \( U \). The result then follows from casting preserving typing.

\[ \square \]

**Lemma A.0.3.** If \((x : U)\) and \((xp : U')\), both well typed, then \(\text{size}(U) \geq \text{size}(U')\), and if \( p \) is non-empty, then \(\text{size}(U) > \text{size}(U')\).

**Proof of Lemma A.0.3.** By induction on \( m \).

Base case: Trivial if \( p \) is the empty spine.

Inductive case: suppose \( p \) is \((p'' u'')\) where \((xp'' : U'')\). Then IH gives \(\text{size}(U) \geq \text{size}(U'')\). If \(xp'' u''\) is well typed, then \(U''\) must be a function type, say \((y : U''1) \rightarrow U''2\), where \( u'' : U''1 \), and \( i \) is the max of the levels of \( U''1 \) and \( U''2 \). Predicativity/universe-hierarchy gives us that \(\text{size}(u'') (i) = 0\), since otherwise \( U''1 \) would need a larger level. The type \( U' \) of \( xp \) is computed by replacing \( u \) with \( u''\); in \( U''2 \), but because \(\text{size}(u'') (i) = 0\), then \(\text{size}(U') (i) = \text{size}(U'') (i)\), i.e replacing \( y \) with \( u'' \) doesn’t introduce any new arrows at level \( i \). So \(\text{size}(U') (i) = 1 + \text{size}(U) (i)\). Then \(\text{size}(U) \geq \text{size}(U'') > \text{size}(U')\).

Looking at SHSubRSpine: our lemma gives that \(\text{size}(U) \geq \text{size}((y : U'1) \rightarrow U'2) > \text{size}(U'_1)\).

We only need to be decreasing in the superscripted type \( U \), since all other recursive references are structurally decreasing. So the recursive references for SHSubRSpine are all on structurally smaller arguments with variable type \( U' \), or on strictly smaller variable types. \( \square \)

**Theorem 3.7.1** (Type safety). If \( \vdash t : U \), then either \( t \rightarrow^* v \) for some \( v \), \( t \rightarrow^* U \), or \( t \) diverges.

**Proof of Theorem 3.7.1.** We prove progress and preservation in the usual way.
For progress, inversion on the typing derivations gives a canonical forms lemma: every closed value is either canonical for its type, $\text{U}$ or $\text{?}$. Every non-value form has a corresponding reduction rule, and the canonical forms lemma guarantees that if non-canonical form is not a redex, then one of its sub-terms can be reduced.

For preservation, most of the reasoning was explained in the chapter. The result holds because of the careful use of evidence: tagging a term with evidence allows it to be assigned any type, so long as the evidence is consistent with both the term’s type and its target type. If building evidence ever fails, then $\text{U}$ is produced, which can be assigned any type. Since the meet always produces a term as precise as its inputs, the evidence applied ensures the reduct has the same type as the redex.

**Theorem 3.7.3.** For any untyped $\lambda$-term $t$ and closing environment $\Gamma$ that maps all variables to type $\text{?}$, then $\Gamma \vdash [t] \Rightarrow \text{?}$. Moreover, if $t$ is closed, then $t \rightarrow^* \nu$ implies that $[t]$ elaborates to $t$ where $t \rightarrow^* \nu$ and stripping evidence from $\nu$ yields $\nu$.

**Proof of Theorem 3.7.3.** First we prove that typing holds.

► **Case Variable**:

Every variable has type $\text{?}$ in $\Gamma$, so the variable synthesizes $\Gamma$

► **Case Abstraction**:

$[\lambda x. t] = \lambda x. [t] :: \text{?}$, and $[t]$ synthesizes $\text{?}$ by IH, so we synthesize type $\text{?}$ by the ascription rule. ► **Case Application**:

Holds by IH and the typing rule for applications, since $\text{?}$ has domain and codomain of $\text{?}$

For the semantics, we can use a simple simulation argument, where each untyped $\beta$-reduction corresponds to a $\beta$-reduction and a removal of an ascription.

**Theorem 3.7.4** (Gradual Guarantee).
\[(\text{Static Guarantee})\] Suppose \(\Gamma \vdash t \in U \text{ and } U \subseteq U'\). If \(\Gamma \subseteq \Gamma'\) and \(t \subseteq t'\), then \(\Gamma' \vdash t' \subseteq U'\).

\[(\text{Dynamic Guarantee})\] Suppose that \(\cdot : t_1 : U, \cdot : t_2 : U', t_1 \subseteq t_1, \text{ and } U \subseteq U'\). If \(t_1 \rightarrow^* t_2\), then \(t_1 \rightarrow^* t_2\) where \(t_2 \subseteq t_2\).

\textit{Proof of Theorem 3.7.4.} The static guarantee is shown by induction on the typing derivation. We omit the details, but the key lemmas are the normalization guarantee (below), that synthesis is monotone in the synthesized type, and the fact that reducing a term’s precision makes it consistent with no less terms.

The dynamic gradual guarantee is proved using a (weak) simulation. That is, for each step \(e_1 \rightarrow e_2\) where \(e_1 \in e_1'\), we show that \(e_1' \rightarrow^* e_2'\) where \(e_2 \in e_2'\). We perform induction on the derivation of \(e_1 \rightarrow e_2\).

First note that if \(e_1'\) is \(\_\) the result holds trivially, likewise if \(e_2\) is \(\top\).

We outline the strategy for the remaining cases. \texttt{StepAppEv,StepAppEvRaw}\n
Then the functions and arguments are precision-related respectively. If the function in \(e_2'\) is a lambda term, then we can \(\beta\)-reduce \(e_2'\). The result then follows from the monotonicity of substitution.

If the function position in \(e_2'\) is \(\_\), then we can step to \(\_\) with \texttt{StepAppDyn}, and our result holds because \(\_\) is the least precise term. \texttt{StepAscr}\n
From the monotonicity of the meet \texttt{StepContext}\n
Then the result holds from IH combined with the fact that precision contains all structural rules. \hfill \(\square\)

\textbf{Lemma 3.7.5} (Normalization Gradual Guarantee). Suppose \(\Gamma_1 \vdash u_1 \leftarrow t_1 \in U_1\). If \(\Gamma_1 \leftarrow \eta \Gamma_2, t_1 \subseteq t_2, \text{ and } U_1 \leftarrow \eta U_2\), then \(\Gamma_2 \vdash u_2 \leftarrow t_2 \subseteq U_2\) where \(u_1 \leftarrow \eta u_2\).

\textit{Proof of Lemma 3.7.5.} Unlike in [58], we do not have separate cases for the boundary between checking and synthesis based on whether precision is increasing or decreasing. The key lemmas are the monotonicity of casting and the meet, both of which are proved with straightforward induction. For casting, any time there is a rule with non-\(\_\) evidence, there is a corresponding rule for \(\_\) evidence, so reducing the precision of evidence to \(\_\) does not cause new failures.

The full result is then obtained by straightforward induction on the normalization derivation, showing that normalization is monotone in both the normal form and, when present, the synthesized type. In particular, \(\_\) synthesizes \(\_\), so
the normal form and synthesized type are both less precise than any other term. Using the body function ensures that reducing the precision of a function in an application produces a less precise result and does not cause failure. □

Lemma 5.4.4 (Precision Reexive). If \( \Gamma \vdash t \leftarrow T \) then \( \Gamma \vdash t \xrightarrow{\text{E}} t \) (For ELABREFL to produce an elaboration that satisfies the \( \xrightarrow{\text{E}} \) side-condition of CASTREFL).

Proof of Lemma 5.4.4. Straightforward induction on the definition of precision. Works because every form has a diagonal rule. □

Lemma 5.4.5 (Composition Safety). If \( t_1 \&_T t_2 \) is not a value and \( \Gamma \vdash t_1 \&_T t_2 \leftarrow T \), then \( t_1 \&_T t_2 \xrightarrow{\text{R}} t_3 \) for some \( t_3 \) and \( \Gamma \vdash t_3 \leftarrow T \) (For progress and preservation)

Proof of Lemma 5.4.5. Each composition of two canonical forms of the same type has a reduction. If one of the composed terms is not a canonical form, then either (1) one of the composed terms can reduce, (2) one term is a \(?_T\) or \(U_T\) where \( T \) is not a function or equality type, and we can reduce with REDCOMPUNK or REDCOMPERR, or (3) one of the composed terms is neutral, and hence the composition is neutral. □

Lemma 5.4.6 (Composition Confluence). If \( t_1 \&_T t_2 \Rightarrow t_3 \) and \( t_1 \&_T t_2 \Rightarrow t'_3 \) maximally, then \( t_3 \Rightarrow t'_3 \), where \( \Rightarrow \) is the parallel reduction relation, standard in confluence proofs [143] (For confluence, which is needed to show that \( \beta \)-reductions preserve types).

Proof of Lemma 5.4.6. The proof is straightforward. We provide the case for composition that fits into the overall induction proof of confluence. If \( t'_3 \) is obtained only by stepping within \( t_1 \) and \( t_2 \) then the result follows from IH. If \( t'_3 \) and \( t'_3 \) are both the results a REDCOMP* rule, then the result follows from IH, plus the preservation of parallel reduction under substitution (since some rules use \&). Finally, if \( t'_3 \) is the result of a REDCOMP* rule but \( t_3 \) is not, then we can step \( t_3 \) with that same rule, and apply the IH and preservation under substitution to get the result. The key is that in each case, there is only one possible non-contextual reduction. □

Lemma 5.4.7 (Composition Lower Bound). If \( \Gamma \vdash t_1 \&_T t_2 \leftarrow T \), then \( \Gamma \vdash t_1 \&_T t_2 \xrightarrow{\text{E}} t_1 \) and \( \Gamma \vdash t_1 \&_T t_2 \xrightarrow{\text{E}} t_2 \) (Preserving the \( \xrightarrow{\text{E}} \) condition of CASTREFL for reduction REDCASTEQ);
Proof of Lemma 5.4.7. Given by PrecComp(L,R).

Lemma A.0.4 (Presynthesis is Monotone). If \( \Gamma \vdash t_1 \Rightarrow^* T_1, \Gamma \vdash t_2 \Rightarrow^* T_2 \) and \( \Gamma_1|\Gamma_2 \vdash t_1 \subseteq alpha t_2 \), then \( \Gamma_1|\Gamma_2 \vdash T_1 \subseteq alpha T_2 \).

Proof of Lemma A.0.4. Straightforward induction on the type derivation. Lemma 5.4.12 is used for CastApp, and Lemma 5.4.13 and the catch-up lemmas are used to show that the results of constrained synthesis are precision related.

Lemma 5.4.8 (Precision Transitive). If \( \Gamma_1|\Gamma_2 \vdash t_1 \subseteq alpha t_2 \) and \( \Gamma_2|\Gamma_3 \vdash t_2 \subseteq alpha t_3 : T \) then \( \Gamma_1|\Gamma_3 \vdash t_1 \subseteq alpha t_3 \) (Preserving the \( \subseteq alpha \) side-condition of CastRefl for reduction RedCastEq);

Proof of Lemma 5.4.8. We prove for by mutual induction for structural and definitional precision. For definitional precision, it follows from the inductive hypothesis, plus confluence.

For structural precision, we proceed by induction on the combined depths of the derivations \( D_1 :: \Gamma_1|\Gamma_2 \vdash t_1 \subseteq alpha t_2 \) and \( D_2 :: \Gamma_2|\Gamma_3 \vdash t_2 \subseteq alpha t_3 \). Cases where both use the same diagonal rule are straightforward, as are any with U on the left or ? on the right. We show a few examples for remaining cases, the reasoning is similar in those we omit.

- **Case** DiagComp, PrecCompL (PrecCompR symmetric):
  
  Then \( t_1 = t_{1L} \& t_1, t_{1R} \), and \( t_2 = t_{2L} \& t_2, t_{2R} \), where \( \Gamma_1|\Gamma_2 \vdash t_{1L} \subseteq alpha t_{2L}, \Gamma_1|\Gamma_2 \vdash t_{1R} \subseteq alpha t_{2R}, \) and \( \Gamma_2|\Gamma_3 \vdash t_{2L} \subseteq alpha t_3 \). IH gives that \( \Gamma_1|\Gamma_3 \vdash t_{1L} \subseteq alpha t_3 \), so PrecCompL yields our result.

- **Case** \( D_1 \) with CastL or \( D_2 \) with CastR:
  
  Follows from IH, plus Lemma A.0.4 to obtain the type precision premises of CastL or CastR.

- **Case** \( D_1 \) with CastR or DiagCast, \( D_2 \) with CastL:
  
  Similar reasoning to above, IH gives us a precision relation between the terms being cast, and then we can apply CastL to obtain result.
**Lemma 5.4.9** (Precision Modulo Conversion). If $\Gamma_1|\Gamma_2 \vdash t_1 \equiv t_2$, where $t_2 \rightarrow^* t'_2$, then $\Gamma_1|\Gamma_2 \vdash t_1 \equiv t'_2$ (Preservation of $\text{CASTREFL}$ under contextual reduction)

*Proof of Lemma 5.4.9.* Immediate from the definition of precision modulo conversion.

**Lemma 5.4.10** (Static Consistency). For any static terms $t_1$ and $t_2$, let $t_1$ and $t_2$ be their embedding in $\text{CASTEQ}$. Then $t_1 \equiv t_2$ iff $t_1 =_{\alpha \beta} t_2$, i.e., if they are statically definitionally equal (for $\text{GEQ}$ to conservatively extend $\text{CIC}$).

*Proof of Lemma 5.4.10.* In the static fragment of $\text{GCIC}$, $\ell$, $\mathcal{U}$ and $\&$ are absent, and all closed equality proofs have the form $\text{refl}(t_w)$ where $t_w$, $t$ and $t'$ are all definitionally equal. So mutual induction on the derivations of $\alpha$-consistency and definitionalse consistency shows the result: rules $\text{CstCompDiag}$, $\text{CstComp}(L,R)$, and $\text{CstUnk}(L,R)$ can never occur. All other rules are a head constructor with consistency premises between the arguments.

**Lemma 5.4.11** (Cast Monotonicity). Suppose that $\Gamma_1|\Gamma_2 \vdash t_1 \equiv t_2$, $\Gamma_1 \vdash t_1 \Rightarrow T_1$ and $\Gamma_2 \vdash t_2 \Rightarrow T_2$ where $\Gamma_1|\Gamma_1 \vdash T_1 \equiv T'_1$ and $\Gamma_2|\Gamma_2 \vdash T_2 \equiv T'_2$. Then $\Gamma_1|\Gamma_2 \vdash \langle T'_1 \Leftarrow T_1 \rangle t_1 \equiv \langle T'_2 \Leftarrow T_2 \rangle t_2$ (For $\text{ELABCST}$ to produce $\equiv$-related elaborations for $\equiv_{\text{Surf}}$-related inputs)

*Proof of Lemma 5.4.11.* Given by $\text{DiagCAST}$.

**Lemma A.0.5** (Substitution Monotone for Structural Precision). Suppose $\Gamma_1|\Gamma_2 \vdash t_1 \equiv_\alpha t_2$, where $\Gamma_1 \vdash t_1 \Rightarrow T_1$ and $\Gamma_2 \vdash t_2 \Rightarrow T_2$. If $\Gamma_1,(x : T_1),\Delta_1|\Gamma_2,(x : T_2),\Delta_2 \vdash t'_1 \equiv_\alpha t'_2$, then $\Gamma_1[t_1/x]|\Delta_1|\Gamma_2[t_2/x]|\Delta_2 \vdash [t_1/x]t'_1 \equiv_\alpha [t_2/x]t'_2$ (For $\text{ELABAPP}$ to be monotone in the return type)

*Proof of Lemma A.0.5.* As Lennon-Bertrand et al. [87] say, the proof follows from weakening of typing, with induction on the precision derivation. The only non-trivial new case is for $\text{PrecCompL}$ ($\text{PrecCompR}$ is symmetric). In this case, we have $\Gamma_1(x : T_1)\Delta_1|\Gamma_2(x : T_2)\Delta_2 \vdash t'_1 \& Tt'_1 \equiv_\alpha t'_2$, where $\Gamma_1(x : T_1)\Delta_1|\Gamma_2(x : T_2)\Delta_2 \vdash t'_1 \equiv_\alpha t'_2$. IH $\Gamma_1[t_1/x]|\Delta_1|\Gamma_2[t_2/x]|\Delta_2 \vdash [t_1/x]t'_1 \equiv_\alpha [t_2/x]t'_2$, so we apply $\text{PrecCompL}$ to get our result.
**Lemma 5.4.12** (Substitution Monotone). Suppose $\Gamma_1 \vdash t_1 \Gamma_2 \Rightarrow t_2$, where $\Gamma_1 \vdash t_1 \Rightarrow T_1$ and $\Gamma_2 \vdash t_2 \Rightarrow T_2$. If $\Gamma_1(x : T_1) \Delta_1 \vdash \Gamma_2(x : T_2) \Delta_2 \vdash t_1' \Gamma_2 \Rightarrow t_2'$, then $\Gamma_1[t_1/x] \Delta_1 \vdash \Gamma_2[t_2/x] \Delta_2 \vdash [t_1/x]t_1' \Gamma_2 \Rightarrow [t_2/x]t_2'$ (For ELABAPP to be monotone in the return type)

**Proof of Lemma 5.4.12.** Follows from Lemma A.0.5 for $\sqsubseteq_a$, plus the preservation of reduction under substitution.

**Lemma A.0.6** (Catch-up Lemmas). Suppose $\Gamma_1 \sqsubseteq_a \Gamma_2$, and $\Gamma_1 \vdash t_1 \sqsubseteq_a t_2$ where $t_1$ has head $h$, and $\Gamma_1 \vdash t_1 \Rightarrow T$. Then $t_2 \rightarrow^* t_2'$ where either (1) $t_2'$ has head $h$ and $\Gamma_1 \vdash t_1 \sqsubseteq_a t_2'$, or (2) $t_2'$ is $?_T$ where $\Gamma_1 \vdash T_1 \sqsubseteq_a T_2$. The same holds for $\sqsubseteq_a$.

**Proof of Lemma A.0.6.** The proof is by induction on the precision derivation, and is identical to gcic, except for the cases regarding equality. First, we note that if $t_1$ has head $h$, it cannot be a composition or a cast expression. The only rules that allow composition on the right require that the term on the left either be a cast or a composition, so $t_2$ cannot be a composition.

**Case** $\text{DiagEQ}$:

Identical to the gcic case for function types.

**Case** $\text{DiagRefl}$:

Then $t_2 = (T_2' \sqsubseteq T_1') \ldots (T_k' \sqsubseteq T_{k-1}' \sqsubseteq) t_2'$, since $\text{CastL}$ and $\text{DiagRefl}$ are the only rules that apply. All of the $T_i'$s are definitionally less precise than $T$, but $T$ must be some $t_L =_{\text{elem}} t_R$ by $\text{CastE Q}$. So each $T_i'$ must either reduce to either $t_L =_{\text{elem}} t_R$ or $?_{\text{Type}}$ (using the previous case for head $\text{Type}$). And the typing premise of CastL, $T_k'$ must be $t_{kL} =_{\text{elem}} t_{kR}$. If $t_k'$ is $?_{kL} =_{\text{elem}} t_{kR}$ (because the first non-cast precision rule was $\text{GENUnk}$) then it reduces with $\text{PropEqUnk}$ to $\text{refl}(t_{kL} \& t_{kR})$. Otherwise it already has the form $\text{refl}(t_{kL} \& t_{kR})$ (because the first non-cast precision rule was $\text{DiagRefl}$). All casts can then be reduced using one of $\text{RedCastEq}$, $\text{RedUpdown}$, $\text{RedEqUnk}$ or $\text{RedEqGerm}$. Since we begin and end with an equality type, the resulting term must have the form $\text{refl}(t_{kL} \& t_{kR})$. 

273
Lemma 5.4.13 (Reduction Monotone). If $\Gamma_1 \vdash t_1 \rightarrow_{\rightarrow} t_2$ and $t_1 \rightarrow^* t_1'$, then $t_2 \rightarrow^* t_2'$ for some $t_2'$ where $\Gamma_1 \vdash t_1' \rightarrow_{\rightarrow} t_1'$ (For DGG, to preserve ELABCST when reducing precision, and to preserve typing under contextual reduction of $\text{refl}(t_w) : \vdash t_1 \equiv t_2$)

Proof of Lemma 5.4.13. We simultaneously prove the result for $\rightarrow$ and $\equiv_{\alpha}$ by mutual induction on the precision derivation. Because PRECCOMP(L,R) have composition on the left-hand side, it is never a possibility in the inversions performed in the GCIC proof. So the GCIC case for each of its reductions still holds for GEQ. We show the cases for the rules added to GEQ. If we ever take a contextual step, the result is immediate from the IH and the corresponding diagonal rule.

**Case** CastL:

The only case now from GCIC is the cast for equality:

**Case** RedCastEq:

Then $t_1 = (t_{\text{endl}} \leftarrow \text{T}_2 \leftarrow \begin{array}{l} \text{repl}(t_w) \vdash t_{\text{startl}} \equiv \text{T}_2 \end{array})$, and $\Gamma_2 \vdash t_2 \rightarrow^* T_2$, where $\Gamma_1 | \Gamma_2 \vdash t_{\text{endl}} \leftarrow \text{T}_2$, $\Gamma_1 | \Gamma_2 \vdash t_{\text{endl}} \leftarrow \text{T}_2$, and $\Gamma_1 | \Gamma_2 \vdash \text{repl}(t_w) \vdash t_{\text{startl}} \equiv \text{T}_2$. Our goal is to show that $\Gamma_1 | \Gamma_2 \vdash \text{repl}(T_2 \leftarrow \begin{array}{l} \text{T}_w \end{array}) \vdash \text{T}_2 \leftarrow \begin{array}{l} \text{Tend} \end{array}$ with $\Gamma_1 | \Gamma_2 \vdash \text{repl}(t_w) \leftarrow \begin{array}{l} t_2 \end{array}$, or (2) $t_2 \rightarrow^? T_2$. The result is immediate for (2) by GENPRECUNK.

For (1), inversion on typing gives that $\text{T}_2 = t_2 \leftarrow \begin{array}{l} t_{\text{endl}} \end{array}$, so inversion on $t_{\text{endl}} \leftarrow \text{T}_2$ gives $\Gamma_1 \vdash t_{\text{endl}} \equiv t_2 \leftarrow \begin{array}{l} T_2 \end{array}$, $\Gamma_1 | \Gamma_2 \vdash t_{\text{endl}} \equiv t_2 \leftarrow \begin{array}{l} T_2 \end{array}$, and $\Gamma_1 | \Gamma_2 \vdash \text{T}_2 \leftarrow \begin{array}{l} T_2 \end{array}$. We get the same for $t_{\text{endl}} \leftarrow \text{T}_2 \leftarrow \begin{array}{l} T_2 \end{array}$. Inversion on the precision from Lemma A.0.6 gives $\Gamma_1 | \Gamma_2 \vdash t_2 \equiv t_2 \leftarrow \begin{array}{l} T_2 \end{array}$, so then CastL gives $\Gamma_1 | \Gamma_2 \vdash \text{T}_2 \leftarrow \begin{array}{l} T_2 \end{array} \equiv \text{T}_2 \leftarrow \begin{array}{l} T_2 \end{array}$.

Two applications of PRECCOMP give $\Gamma_1 | \Gamma_2 \vdash \text{T}_2 \leftarrow \begin{array}{l} T_2 \end{array}$, so then our result is built with DIAGREFL.

**Case** PRECCOMP (PRECCOMP is symmetric):
Then $t_1 = t_{1L} \& t_{1R}$, with $\mathcal{D} :: \Gamma_1|\Gamma_2 \vdash t_{1L} \sqsubseteq_\alpha t_2$. We proceed by cases on the reduction rule used on $t_{1L} \& T t_{1R}$.

**Case RedCompgGerm**:

Then $t_{1L} = \langle \text{Type}_r \leftarrow \text{germ}(h) \rangle t'_{1L}$ and $t_{1R} = \langle \text{Type}_r \leftarrow \text{germ}(h) \rangle t'_{1R}$. We proceed by inversion on $\mathcal{D}$.

**Case DiagCast**:

Then $t_2 = \langle T_{\text{2end}} \Leftarrow T_{\text{2start}} \rangle t'_2$, with $\Gamma_1|\Gamma_2 \vdash t'_{1L} \sqsubseteq_\alpha t'_2$, $\Gamma_1|\Gamma_2 \vdash \text{germ}(h) \sqsubseteq_\alpha T_{\text{2start}}$ and $\Gamma_1|\Gamma_2 \vdash \text{Type}_r \sqsubseteq_\alpha T_{\text{2end}}$. Then $T_{\text{2end}}$ is $\text{Type}_r$, with some number of casts, so we can apply GenUnk and CastL to get $\Gamma_1|\Gamma_2 \vdash \text{germ}(h) \sqsubseteq_\alpha T_{\text{2end}}$. So by PrecCompL we have $\Gamma_1|\Gamma_2 \vdash t'_{1L} \& \text{germ}(h) t'_{1R} \sqsubseteq_\alpha t'_2$. Then by DiagCast we have our result.

**Case CastL**:

Then we have $\Gamma_1|\Gamma_2 \vdash t'_{1L} \sqsubseteq_\alpha t_2$, so by PrecCompL we have $\Gamma_1|\Gamma_2 \vdash t'_{1L} \& \text{germ}(h) t'_{1R} \sqsubseteq_\alpha t_2$. The result then follows from CastL.

**Case RedCompUnkL**:

Then $t_2 \rightarrow^* ?_T$, giving our result.

**Case RedCompUnkR**:

Follows from premise of PrecCompL.

**Case RedCompErr, RedCompHeadErr, RedCompGermErr**:

Trivial since $\mathcal{U}$ is least.

**Case RedCompEq, RedCompRefL, RedCompInd, RedCompLam, RedCompCon, RedCompPi**:

In each case, $t_2$ has the same head as $t_{1L}$ and $t_{1R}$, so we can push the use of PrecCompL deeper, applying the necessary diagonal rules, along with Lemma 5.4.12.
Case \textbf{DiagComp}:

By cases on the reduction rule used on the left.

\begin{itemize}
\item \textbf{Case RedCompGerm}:
\end{itemize}

Then we have $\Gamma_1 | \Gamma_2 \vdash \langle \text{Type}_r \iff \text{germ}(h) \rangle t_L \& \text{Type}_r, \langle \text{Type}_r \iff \text{germ}(h) \rangle t_L \subseteq \alpha$

$t_{2L} \& t_{2R}$. Then by Lemma A.0.6 $T_2$ must reduce to $\text{Type}_r$, since it is less precise than $\text{Type}_r$. The precision relation between $\langle \text{Type}_r \iff \text{germ}(h) \rangle t_L$

and $t_{2L}$ must either use DiagCast or CastL, likewise for on the RHS. If both use DiagCast, then we can step $t_{2L} \& t_{2R}$ with RedCompGerm and use DiagCast and DiagComp for our result.

Consider then the case where the precision relation with $t_{2L}$ uses CastL, and with $t_{2R}$ uses DiagCast (the case for vice versa is symmetric). We will get our result in zero steps of the RHS.

Here, $\Gamma_1 | \Gamma_2 \vdash t_{1L} \subseteq \alpha t_{2L}$, with and $t_{2R} = \langle T_{2end} \iff T_{2start} \rangle t_{2R}'$

with $\Gamma_1 | \Gamma_2 \vdash t_{1R} \subseteq \alpha t_{2R}'$, $\Gamma_1 | \Gamma_2 \vdash \text{germ}(h) \subseteq \alpha T_{2start}$ and $\Gamma_1 | \Gamma_2 \vdash \text{Type}_r \subseteq \alpha T_{2end}$. Then $T_{2end}$ is $\text{Type}_r$ with some number of casts, so we can apply GenUNK and CastL to get $\Gamma_1 | \Gamma_2 \vdash \text{germ}(h) \subseteq \alpha T_{2end}$. Then by CastR, we have $\Gamma_1 | \Gamma_2 \vdash t_{1R} \subseteq \alpha \langle T_{2end} \iff T_{2start} \rangle t_{2R}'$. So by DiagComp we have $\Gamma \vdash t_{1L} \& \text{germ}(h) t_{1R} \subseteq \alpha t_{2L} & T_{2end} \langle T_{2end} \iff T_{2start} \rangle t_{2R}'$. Finally, our result comes from applying CastL. The case where both use CastL is similar, but without the need to use CastR.

\begin{itemize}
\item \textbf{Case RedCompUnk}:
\end{itemize}

We handle RedCompUnk(L), the other case is symmetric. In this case, we have $\Gamma_1 | \Gamma_2 \vdash ?_{T_1} \& T_1, t_{1R} \subseteq \alpha t_{2L} \& T_2$. Inversion gives that $t_{2L}$ must be some sequence of casts applied to $?_{T_2}'$, where each cast type and $T_2'$ are all less precise than $T_1$. Because $T$ is one of Type\(r\), C(→) or ?Type\(r\), and because each cast type is less precise, $t_{2L} \rightarrow^* ?_{T_1}$. So we can step with RedCompUnk(L) and get our result from inversion on the original precision derivation.

\begin{itemize}
\item \textbf{Case RedCompErr, RedCompHeadErr, RedCompGermErr}:
\end{itemize}

276
Trivial since $U$ is least.

▶ **Case** RedCompEq, RedCompRefl, RedCompInd, RedCompLam, RedCompCon, RedCompP1:

Then we have $\Gamma_1 | \Gamma_2 \vdash t_{1L} &_T t_{1R} \subseteq_\alpha t_{2L} &_T t_{2R}$, where $t_{1L}$ and $t_{2L}$ have the same head $h$. We have $t_{1L} &_T t_{1R} \rightarrow^* t'_1$ and want to find $t'_2$ such that $t_{2L} &_T t_{2R} \rightarrow^* t'_2$ and $\Gamma_1 | \Gamma_2 \vdash t'_1 \subseteq_\alpha t'_2$. Lemma A.0.6 on each side of $\&$ gives $t_{2L} &_T t_{2R} \rightarrow^* t'_{2L} &_T t'_{2R}$ that is less precise than $t_{1L} &_T t_{1R}$, where $t'_{2L}$ and $t'_{2R}$ both have head $h$. This term then steps with the same reduction as $t_{1L} &_T t_{1R}$, and using the appropriate diagonal rules and the monotonicity of substitution gives our result.

▶ **Case** DiagJ:

Then the step is with RedJ. So we can step the RHS with RedJ, and obtain our result with DiagCAST and the monotonicity of substitution.

▶ **Case** DiagEQ:

Then only a contextual step is possible, and the result follows from IH, since the type-comparisons in DiagEQ are all modulo reduction.

▶ **Case** DiagRefl:

Then only a contextual step is possible, so we take the same step in the RHS and use DiagRefl for our result.

Lemma A.0.7 ($\alpha$-Consistency Monotone for Precision). If $\Gamma \vdash t_{l_{owL}} \subseteq_\alpha t_{h_{ighL}}$ and $t_{l_{owL}} \equiv_\alpha t_{R}$, then $t_{h_{ighL}} \equiv_\alpha t_{R}$

*Proof of Lemma A.0.7.* By induction on the derivations of $\Gamma_1 | \Gamma_2 \vdash t_{l_{owL}} \subseteq_\alpha t'_{h_{ighL}}$.

▶ **Case** $\equiv_\alpha$ derived with CstCompR or CstCASTR:
If some sequence of \(C\)/\(s\)/\(c\)/\(t\)/\(c\)/\(o\)/\(m\)/\(p\)/\(R\) and \(C\)/\(s\)/\(c\)/\(t\)/\(c\)/\(a\)/\(s\)/\(c\)/\(t\)/\(R\) was used, we can unwrap the derivation from these until there is a use of a rule that constrains the syntax of the LHS. We then use the derivation from the corresponding case below, then re-apply the same sequence of \(C\)/\(s\)/\(c\)/\(t\)/\(c\)/\(o\)/\(m\)/\(p\)/\(R\) and \(C\)/\(s\)/\(c\)/\(t\)/\(c\)/\(a\)/\(s\)/\(c\)/\(t\)/\(R\) to obtain our result.

**Case** \(\text{PrecComp}\ L\) (\(\text{PrecComp}\) is symmetric):

So \(t_{\text{low}L} = t_1 \&_T t_2\), and \(\Gamma_1 \mid \Gamma_2 \vdash t_1 \sqsubseteq_\alpha t_{\text{high}L}\). Then by IH we have \(t_{\text{high}L} \equiv_\alpha t_R\).

**Case** \(\text{DiagComp}\):

Then \(t_{\text{low}L} = t_1 \&_T t_2\) and \(t_{\text{high}L} = t_1' \&_T t_2'\), where \(\Gamma_1 \mid \Gamma_2 \vdash t_1 \sqsubseteq_\alpha t_1'\) and \(\Gamma_1 \mid \Gamma_2 \vdash t_2 \sqsubseteq_\alpha t_2'\). If \(\equiv_\alpha\) was derived with \(\text{CstCompDiag}\), inversion gives \(t_R = t_{R1} \&_T t_{R2}\), so we can apply IH and \(\text{CstCompDiag}\). If \(\equiv_\alpha\) was derived with \(\text{CstCompL}\), we can use IH and \(\text{CstCompL}\).

**Case** \(\text{DiagRefl}\):

Follows from IH plus \(\text{CompRefl}\).

**Case** Remaining cases:

Same as \(\text{Gcic}\)

\(\square\)

**Lemma A.0.8** (\(\alpha\)-Consistency Symmetric). \(\equiv_\alpha\) is symmetric.

*Proof of Lemma A.0.8.* Straightforward induction. Rules \(\text{CstVar}\) and \(\text{CstType}\) have identical terms on each side. Any uses of \(\text{CstCompL}\) can be turned into \(\text{CstCompR}\) and vice versa, likewise for \(\text{CstCastL}\) and \(\text{CstCastR}\). The other cases follow easily from IH. \(\square\)

**Lemma 5.4.14** (Consistency Monotone for Precision). If \(\Gamma \mid t_1 \sqsubseteq_{--} t_1'\) and \(\Gamma \mid t_2 \sqsubseteq_{--} t_2'\), and \(t_1 \equiv_{--} t_2\), then \(t_1' \equiv_{--} t_2'\) (So reducing precision of \(V\) and \(V'\) preserves \(\text{ElabCst}\)).

*Proof of Lemma 5.4.14.* Follows from Lemma A.0.8, Lemma A.0.7 and Lemma 5.4.13. \(\square\)

278
Lemma 5.4.15 (Structural Precision). $\xrightarrow{\text{Structural Precision}}$ contains all structural rules (For homomorphic elaboration rules to produce $\xrightarrow{\text{Surf-related elaboration}}$ related elaboration for $\xrightarrow{\text{Surf-related inputs}}$)

Proof of Lemma 5.4.15. Given by the diagonal rules of precision. $\square$

Lemma A.0.9 (Canonical Forms). If $\Gamma \vdash v \iff T$ or $\Gamma \vdash v \Rightarrow T$, and $v$ is not neutral, then $v$ is either $?T_0$ or $?T$, where $T'$ is convertible with $T$, or one of the following holds:

- $T = (x : T_1) \rightarrow T_2$ and $v = \lambda(x : T_1). t$
- $T = t_1 =_{T'} t_2$ and $v = \text{refl}(t_w), t'_1 \equiv t'_2$
- $T = C(T)$ and $v = D^C(t')$
- $T = \text{Type}_\ell$ and $v = (x : T_1) \rightarrow T_2$ or $v = C(T)$ or $v = t_1 =_{T'} t_2$ or $v = \text{Type}_{\ell'}$ for $\ell' < \ell$.

Proof of Lemma A.0.9. By induction on the typing derivation.

- **Case** CastCheck:
  - Follows from IH on the synthesis derivation

- **Case** CastVar, CastApp, CastMatch, CastCast, CastJ, CastComp:
  - If $v$ is not neutral, then it cannot be a value, since the only value forms for eliminations are neutrals.

- **Case** CastFun, CastType, CastFun, CastApp, CastInd, CastCtor, CastUnk, CastErr, CastEq, CastRefl:
  - Immediate from the form of the typing derivation.

$\square$

Lemma 5.4.16 (Confluence, Progress, Preservation and Elaboration). The following hold:

- $\rightarrow$ is confluent.

279
• If $\Gamma \vdash t \iff T$, then $t$ is a value or $t \rightarrow t'$ for some $t'$.

• If $\Gamma \vdash t_1 \iff T$ and $t_1 \rightarrow t_2$ then $\Gamma \vdash t_2 \iff T$.

• If $\Gamma \vdash t \rightarrow t \iff T$, then $\Gamma \vdash t \iff T$.

Proof of Lemma 5.4.16. (1) As in [87]. The proof uses the usual parallel reduction strategy. The additional cases are for $\text{REDJ}$, $\text{REDCASTEQ}$, $\text{PROPEQUNK}$ and $\text{PROPEQEQ}$, all of which are straightforward, and for $\&$, which is given by our criteria.

(2) As in [87]. We perform mutual induction on the typing derivation with the corresponding propositions for synthesis and constrained synthesis. The gradual criteria guarantee that the composition does not interfere with the canonical-forms property for functions, types (in cast statements) and members of inductive types, so all cases from $\text{GCIC}$ are readily adapted to $\text{CASTEQ}$. Additional cases are for $\text{CASTUNK}$, $\text{CASTERR}$, $\text{CASTEQ}$, $\text{CASTREFL}$, $\text{CASTJ}$, and possibly $\&$. For $\text{CASTUNK}$ and $\text{CASTERR}$, there is a new case because of the new type former $=$, but we always step with $\text{PROPEQUNK}$ or $\text{PROPEQErr}$. For $\text{CASTEQ}$ and $\text{CASTREFL}$, by our hypothesis, either we can take a step in a sub-term or all sub-terms are normal, meaning the entire term is normal. For $\text{CASTJ}$, the arguments to $J$ must all be normal, or a step can be taken in one of them. If they are all normal, then the canonical forms lemma gives that the scrutinee must be $\text{refl}$, enabling a reduction with $\text{REDJ}$. Our gradual criteria ensure that any non-value $\&$ terms can step.

(3) As in [87]. We perform mutual induction on the typing derivation with the corresponding propositions for synthesis and constrained synthesis. The new cases are for $\text{REDJ}$, $\text{REDCASTEQ}$, $\text{PROPEQUNK}$ and $\text{PROPEQErr}$, as well as $\&$. For $\text{REDJ}$, the cast types are well typed by preservation of typing under substitution, and the whole expression can be typed with two applications of $\text{CASTCAST}$. For $\text{REDCASTEQ}$, the witness and cast-types are well typed by the premise, so casting the witness is well typed at the destination type by $\text{CASTCAST}$. Our gradual criteria give that composing terms of the same type preserves that type. Finally, we have that the witness is more precise than both endpoints, by transitivity and the lower-bound property from the gradual criteria, allowing us to apply $\text{CASTREFL}$. The cases for $\text{PROPEQUNK}$ and $\text{PROPEQErr}$ are trivial, and the case for $\&$ follows from the gradual criteria. Finally, we have contextual steps. These are
all straightforward, or identical to GCIC, except for preserving the precision side-
conditions of CASTREFL, where it follows from our definition of precision modulo
convertibility.

(4) As in [87]. We perform mutual induction on the typing derivation with
the corresponding propositions for synthesis and constrained synthesis. The new
cases are ELABEQ, which is trivial, along with ELABREFL. For ELABREFL, our crite-
rion give reflexivity of $\downarrow$, so this combined with our premises allow us to apply
CASTREFL to type the elaboration.

Corollary 5.4.18 (Weak Canonicity). Suppose $\vdash t : V$. Then either $t$ diverges, or
$t \rightarrow^{*} v$ where $v$ is $?_V$ or $\uparrow_V$, or the following hold:

- If $V$ is $\lambda x : T_1 \rightarrow T_2$ then $v$ is $\lambda x. t'$
- If $V$ is $C_{\downarrow(i)}(t_1)$ then $v$ is $D_{\downarrow(i)}(t_2)$ for some D.
- If $V$ is $t_1 =_T t_2$ then $v$ is refl($t'$)$_{t_1=t_2}$
- If $V$ is $\text{Type}_i$ then $v$ is one of $C_{\downarrow(i)}(t_1), (x : T_1) \rightarrow T_2, \text{Type}_{i-1}$ or $t_1 =_T t_2$.

Proof of Corollary 5.4.18. Follows from Lemma A.0.9 with type safety.

Theorem 5.4.19 (Conservativity). For any bcic-terms $t$ and $T$, let $t$ and $T$ be the
GEQ terms corresponding to $t$ and $T$ by mapping bcic $\lambda$ to GEQ $\lambda$, etc. Then $\vdash t \iff T$
tt.

Proof of Theorem 5.4.19. BCIC as given by Lennon-Bertrand [86] does not have proposi-
tional equality, but it is easily added with the types given in Fig. 5.2. We identify
refl : $t =_T t$ with refl($t'$)$_{t_1=t_2}$.

The “only if” direction is straightforward, since each BCIC rule has an analogue
in GEQ. For the other direction, we need to transform each GCIC type derivation
into a BCIC derivation. However, this is straightforward induction: all typing rules
have corresponding BCIC rules, except for CASTCOMP, CASTUNK and CASTERR, but
these all use gradual features not present in the embedding of a GCIC term into
BCIC. For CASTCHECK, the result follows from Lemma 5.4.10.

Lemma 5.4.20 (Elaboration Gradual Guarantee). Suppose $t_1 \inSurf t_2$ and $\Gamma_1 \subseteq \Gamma_2$
(i.e. entries in $\Gamma_1$ and $\Gamma_2$ are respectively related by $\subseteq$). Then:
• If $\Gamma_1 \vdash t_1 \rightarrow t_1 \Leftarrow T$ then $\Gamma_2 \vdash t_2 \rightarrow t_2 \Leftarrow T$ for some $t_2$ where $\Gamma_3 | \Gamma_2 \vdash t_1 \Downarrow t_2$.

• If $\Gamma_1 \vdash t_1 \rightarrow T_1$ then $\Gamma_2 \vdash t_2 \rightarrow T_2$ for some $T_2$, $t_2$ where we have $\Gamma_3 \vdash T \Downarrow \text{Type}_T$ and $\Gamma_1 | \Gamma_2 \vdash t_1 \Downarrow t_2$.

Proof of Lemma 5.4.20. Our gradual criteria ensure that every rule of structural precision from [87] is admissible for $\Downarrow$, so our proof follows the same format. The new cases are for ELABEQ, and ELABREFL, which are immediate from the IH. We reiterate the case for ELABCST to show how it fits with the gradual criteria.

Suppose that $\Gamma_1 \vdash t_1 \rightarrow T_1$, that $\Gamma_2 \vdash t_2 \rightarrow T_2$, and that $t_1 \in \text{Surf} t_2$. We know that $T \Downarrow \text{Type} T_1$, and our goal is to show that $\Gamma_3 | \Gamma_2 \vdash (T_1 \Downarrow) T t_1 \Downarrow (T_2 \Downarrow) T t_2$ and that $T \Downarrow \text{Type} T_2$. The former is immediate from Lemma 5.4.11. The latter follows from Lemma 5.4.14. □

Lemma 6.7.1 (Soundness). If $\Gamma \vdash t \Leftarrow T$ and $t \sim_{E} t'$ then

$$\left( x_k : \mathcal{L}_a \Gamma \left[ \Gamma(x_k) \right] \right)^k \rightarrow (E \left[ t \Downarrow \right], E \left[ t' \Downarrow \right])$$

is inhabited in $\text{TCCTT}$. Moreover, $E \left[ t \Downarrow \right]$ is reducible.

Proof of Lemma 6.7.1. Each eliminator form is reducible because it corresponds to a pattern-matching definition, so reducing the entire match to the corresponding branch (or equivalently, the proper argument given to an inductive eliminator) is a reduction step. Likewise, for function application, taking the second element of the produced pair is a reduction step.

In each case the equality is definitional, except for casting functions to and from $\forall \text{Ty}$, which are blocked by transporting across $\text{Eq}$. ▶ Case Propagation reductions:

- For computing $\forall$ and $\forall$, the result holds from our definition of $\forall$From and $\forall$From: we $\eta$-expand at function types and compute the meet of the endpoints for equality. For eliminating or casting $\forall$ or $\forall$ of an inductive type, we produce $\mathcal{W}$ or $\mathcal{W}$, which is equal to the translations of $\forall$ and $\forall$ respectively.

- Case RedTagUpDown:
headMatch and fromGerm are defined to directly mirror the behavior of FromGerm.

**Case** RedTAGUpDOWNIND:

The reduction rule was designed specifically to mirror the behavior of the model.

**Case** Other casting rules, composition rules:

In each case, reducing the definition of cast on the translation of the redex gives exactly the translation of the reduct. One important detail is that the model constructs terms using the monadic bind, just like cast does, so even in the exact case equal computations are produced.

**Case** REDJ:

The translation of J performs precisely the cast specified by the reduction rule. The presentation omitted it for clarity, but in the Agda model we wrap witnesses in a wrapper type, which ensures that computing such a cast pattern matches on the equality proof, so it invokes the reduction we need for our termination argument.

Lemma 6.7.2 (Model Preserves Types). If \( \Gamma \vdash t \equiv T \) then

\[
E[[t]]_{\alpha} : (x_k : L_{\alpha} \Rightarrow T[[\Gamma(x_k)]] \rightarrow (L_{\alpha} \Rightarrow T[[T]]) \text{ in } \text{tctt}.
\]

**Proof of Lemma 6.7.2.** By induction on the typing derivation. The approach is standard: typedness of subterms derives from the hypothesis. We give the general strategy for each case.

**Case** CastCHECK, constrained synthesis:

Follows from soundness

**Case** CastVAR:

Trivial

**Case** CastTYPE, CastFUN:
El $C (\ell + 1)$ is $C \ell$, and the corresponding types both translated to codes in $C \ell$

**Case CastApp** :
Application takes the exact component of the function, which has the desired type (the codomain applied to the approximation of the argument, under the $L_\alpha$ monad).

**Case CastAbs** :
The translation for a lambda term produces a function producing a pure approximate and monadic exact result, which aligns with El $(C \Pi \text{dom cod})$

**Case CastInd** :
The result is a code, which matches the expected type, and we assume descFor was provided matching the parameter and index types.

**Case CastEq** :
The approximate translations of the equated terms produces a result that, when extracted with from$L$, yields the element type for the given code. The entire expression produces a member of $C \ell$, which matches El CType

**Case CastCtor,CastMatch** :
We omit the details, but follow the approach of Diehl and Sheard [48]

**Case Remaining rules** :
The elimination operations, along with $\? \text{ and } U$, produce the elements of the given code, under $L_\alpha$, which is precisely the expected type of each operation.

□