REDEX-PLUS: A Metanotation for Programming Languages

by

Junfeng Xu

B.Eng., Imperial College London, 2019

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

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

The University of British Columbia

(Vancouver)

August 2023

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

**REDEX-PLUS:**
**A Metanotation for Programming Languages**

submitted by *Junfeng Xu* in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

**Examine Committee:**
William Bowman, Professor, Computer Science, UBC  
*Supervisor*

Ronald Garcia, Professor, Computer Science, UBC  
*Supervisory Committee Member*
Abstract

When defining the syntax and semantics of programming languages, computer scientists often use some kind of computer science metanotations. These metanotations are often informal or under-defined: human readers can understand them intuitively, but they are unsuitable for use in rigorous formal proofs. The aim of this project is to develop a tool that bridges the gap between the intuitive metanotations for defining programming languages, and proof assistant languages in which properties of the languages defined can be formally proven.

In this thesis, I present REDEX-PLUS, an implementation of a computer science metanotation based on the Racket language Redex, which is used to define programming language models. REDEX-PLUS is a compiler that translates language models defined in the Redex language to corresponding definitions in various proof assistant languages. REDEX-PLUS also generates boilerplate such as binding operations, saving the users from the tedious job of writing boilerplate by hand. REDEX-PLUS supports Coq, Agda, SMT-LIB, and Beluga as translation targets; it can faithfully translate language models with multiple types of bound variables; and it gives users the choice between multiple ways to represent variable binding when using the Coq or Agda targets. I evaluate REDEX-PLUS by formalizing a variant of Simply Typed Lambda Calculus and proving some important properties of the language model in both Coq and Agda, and by formalizing a variant of System F and attempting to prove the same properties in Coq.

The source code repository of REDEX-PLUS is publicly available at https://gitlab.com/xujunfeng/redex-plus, and on Zenodo [41].
Lay Summary

When reasoning about programming languages, computer scientists use *metanotations* to state how the programming languages look and behave. However, these metanotations are often informal, and cannot be used in formal, rigorous, mathematical proofs. In this thesis I present REDEX-PLUS, a tool that translates programming language definitions written using a certain metanotation into code that can be used in formal proofs. I also demonstrate REDEX-PLUS’s usefulness by using it to translate the definition of a certain well-known programming language into code, and then using the translated code to prove some important properties of the programming language in question.
Preface

This thesis presents original, unpublished work by the author Junfeng Xu under the supervision of Professor William Bowman.

REDEX-PLUS, the tool presented in this thesis, is based on the Racket library Redex. The SMT-LIB translation target (see section 4.2.3) was developed as the author’s personal project for the CPSC 513 course. The proofs in section 7.2 are based on similar proofs in the books Programming Language Foundations in Agda and Software Foundations.
# Table of Contents

Abstract ......................................................... iii

Lay Summary ....................................................... iv

Preface ............................................................ v

Table of Contents ............................................... vi

List of Tables .................................................... ix

List of Figures ................................................... x

Acknowledgments ................................................ xiv

1 Introduction: Who Formalizes the Formalization? .......... 1
  1.1 “The Most Popular Programming Language” ............... 1
  1.2 Dreaming of a Computer Science Metanotation .......... 2
  1.3 REDEX-PLUS ................................................. 4
     1.3.1 Structure of this Thesis ............................ 5

2 Background: Existing Metanotation Implementations ....... 6
  2.1 Ott .......................................................... 7
     2.1.1 LNgen ................................................ 7
  2.2 Lem ........................................................ 8
  2.3 Spoofax ..................................................... 9
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td><strong>Redex-Plus: A Computer Science Metanotation</strong></td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Redex</td>
<td>10</td>
</tr>
<tr>
<td>3.1.1</td>
<td>A Basic Example</td>
<td>12</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Problems with Translating Redex</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>The Supported Redex Subset</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td><strong>Translating Language Models, and Generating Code</strong></td>
<td>22</td>
</tr>
<tr>
<td>4.1</td>
<td>Language Models to Intermediate Representation</td>
<td>23</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Poultry: The Intermediate Representation</td>
<td>23</td>
</tr>
<tr>
<td>4.1.2</td>
<td>From Redex To Poultry</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Generating Code</td>
<td>35</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Dependent Typing Proof Assistants</td>
<td>36</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Beluga</td>
<td>40</td>
</tr>
<tr>
<td>4.2.3</td>
<td>SMT-LIB</td>
<td>41</td>
</tr>
<tr>
<td>5</td>
<td><strong>Variable Binding: Representation &amp; Implementation</strong></td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td>Named</td>
<td>44</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Determining when to Generate Binding Operations</td>
<td>47</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Variable Capturing</td>
<td>47</td>
</tr>
<tr>
<td>5.2</td>
<td>Locally Nameless</td>
<td>49</td>
</tr>
<tr>
<td>5.3</td>
<td>Shifted Names</td>
<td>52</td>
</tr>
<tr>
<td>5.4</td>
<td>Higher-Order Abstract Syntax</td>
<td>54</td>
</tr>
<tr>
<td>6</td>
<td><strong>Implementation</strong></td>
<td>57</td>
</tr>
<tr>
<td>6.1</td>
<td>Interface</td>
<td>57</td>
</tr>
<tr>
<td>6.2</td>
<td>Internal Workings</td>
<td>58</td>
</tr>
<tr>
<td>6.3</td>
<td>The Compilation Passes</td>
<td>59</td>
</tr>
<tr>
<td>6.4</td>
<td>Adding New Translation Targets</td>
<td>62</td>
</tr>
<tr>
<td>7</td>
<td><strong>Evaluation</strong></td>
<td>65</td>
</tr>
<tr>
<td>7.1</td>
<td>Automatic Code Generation</td>
<td>65</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Discussion</td>
<td>67</td>
</tr>
<tr>
<td>7.2</td>
<td>Properties of the Simply Typed Lambda Calculus</td>
<td>68</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Progress</td>
<td>72</td>
</tr>
</tbody>
</table>
List of Tables

Table 6.1 Compilation passes for all REDEX-PLUS statements and targets. “○” means this pass is a part of the translation process for the target; “|” means it is not; “△” means it depends on user setting; “⊕” means only one of the passes marked as so is included, depending on user settings; and “×” means this statement is not supported by the target. The targets (Coq, Agda, Beluga, and SMT-LIB) are abbreviated to their initial letters.

Table 7.1 Breakdown of the amount of generated code, in terms of lines of code.
# List of Figures

| Figure 3.1 | The Redex language definition for the F language. | 13 |
| Figure 3.2 | Additional metafunction and judgement form definitions used to define the typing relation in the F language. | 14 |
| Figure 3.3 | The typing judgement and reduction relation for the F language. | 15 |
| Figure 3.4 | Racket code for generating the list of derivations (ds) and displaying them. Only one derivation is shown as in this example there is only one possible derivation. | 16 |
| Figure 3.5 | Pattern language supported by REDEX-PLUS. | 19 |
| Figure 4.1 | The Tree language model. | 22 |
| Figure 4.2 | Using for-language to separate statements belonging to different Redex languages. | 24 |
| Figure 4.3 | Defining a binary tree type with define-types. | 26 |
| Figure 4.4 | The Redex metafunction that mirrors a binary tree. | 26 |
| Figure 4.5 | The generated IR of the statement in figure 4.4, with all arguments annotated. All user-defined metafunctions will have the language name prefixed to the function name. | 27 |
| Figure 4.6 | The Redex metafunction that compares two binary trees. The first case, which returns true, is only matched when both arguments are syntactically the same. If they are not, then the function falls back to the second case and returns false. | 28 |
| Figure 4.7 | The IR of the same-tree function definition. Notice the #:if expression inside the top level #:match, in which the two arguments are compared. | 29 |
Figure 4.8 The function body of \texttt{same-tree} after simplification.

Figure 4.9 The Redex judgement representing the “less-than” relation between Peano numbers.

Figure 4.10 The generated IR of the statement in figure 4.9. The first two arguments of \texttt{define-proposition} declares the name and signature, in terms of type names, of the judgement. A list of rules follows, in which each rule contains a list of (implicitly) bound variables used in the rule, the judgement constructed by the rule, and a potentially empty (in the case of axioms) list of premises. The \texttt{#:term} keyword helps the later translation process distinguish judgement forms from terms.

Figure 4.11 The Redex definition of $\beta$-reduction in System F.

Figure 4.12 The function definition corresponding to the \texttt{subst} reduction case. The argument type (\texttt{#:well-bound e}) means an $e$ that has a type-level guarantee of its well-boundedness. The well-boundedness requirement is only relevant when using binding representations that can express well-boundedness, such as shifted names. When using other binding representations, the requirement is ignored.

Figure 4.13 $\beta$-reduction of System F represented as a proposition.

Figure 4.14 Compatible closure of $\beta$-reduction of System F represented as a proposition; abridged.

Figure 4.15 The Coq definition of a binary tree generated by \texttt{REDEX-PLUS}.

Figure 4.16 The “less than” relation defined in figure 4.9 translated into Coq, and the associated \texttt{autoderive} definitions.

Figure 4.17 Proving that a natural number is less than itself plus two by induction on natural numbers. Both induction cases can be trivially solved by \texttt{autoderive}.

Figure 4.18 The Agda definition of a binary tree, as well as the “less than” relation between Peano numbers.

Figure 4.19 System F represented in Beluga. The natural number type \texttt{nat} is defined in the prelude.
| Figure 7.1 | Breakdown of the amount of generated code, in terms of lines of code. | 67 |
| Figure 7.2 | The Redex language definition of STLC and $\beta$-reduction. | 68 |
| Figure 7.3 | The Redex definition of the typing judgement in STLC. | 70 |
| Figure 7.4 | Notation definitions for STLC, in Coq and Agda. | 71 |
| Figure 7.5 | Definitions of $\beta$-reduction and the associated notation in Coq and Agda. The rule for reducing the body of an lambda abstraction has been removed, and the left sub-expression of a function application is not required to be a value before the right sub-expression could reduce. | 74 |
| Figure 7.6 | The Redex definition of the a simple arithmetic expression language and its reduction relation. | 78 |
| Figure 7.7 | SMT-LIB code for proving two reduction relations. The “push” and “pop” commands are for delimiting each of the two proofs. | 79 |
| Figure 7.8 | SMT-LIB code for finding what the expression (add 1 1) reduces to, and the Z3 output from running the code. Notice that the model Z3 came up with states that Arith-eval-single-step is always true (i.e. every term reduces to every other term), and Arith-eval is true as long as the reduction result is zero or two, which are both clearly nonsensical. | 80 |
| Figure 7.9 | The Redex definition of natural numbers and two relations. | 81 |
| Figure 7.10 | SMT-LIB code for checking the correctness of two simple theorems about natural numbers. | 81 |
Acknowledgments

First and foremost, I would like to thank Professor William Bowman, my supervisor, for guiding me through not only the REDEX-PLUS project but also my life as a graduate student.

I would also like to thank Professor Ronald Garcia for his valuable comments on this thesis.

Last but not least, I would like to thank Paulette Koronkevich for her advice on writing proofs, and to thank Professor Margo Seltzer, and my friends Emily, Fangyi Zhou, and Molly, for their support throughout my journey.
Chapter 1

Introduction

Who Formalizes the Formalization?

1.1 “The Most Popular Programming Language”

To study programming languages (PL), we often first formalize them. Usually, this means using a terse, unambiguous abstract notation to represent an existing language, or to define a new language. Then, we give this abstract notation a meaning using a set of well-defined mathematical rules.

To do so, researchers rely on some metanotations: the notations used to define and specify programming languages. These metanotations are not monolithic products of some conscious design process. Instead, they are conglomerates consisting of many notations with origins in computer science and logics. For example, Backus-Naur Form (BNF), originally used to describe the concrete syntax of ALGOL [4], is commonly used for defining the abstract syntax of languages; and Gentzen’s syntax for derivation rules [22] is used for inferring properties of language terms.

Despite the diversity of the metanotations, they are used to accomplish similar sets of goals. In existing PL literature and related tools, people use these metanotations to:

- define a formal syntax of a language, with either BNF, or more commonly,
one of the extended variants of BNF.

- designate some terms as “variables”, and to safely substitute a variable with a term;

- define a reduction semantics for the language, that describes how one language term can be “reduced” or “evaluated” to another. Often, the order of reduction is also defined; and

- reason about and prove properties of terms, usually by incorporating the language into a logical representation, such as first-order logic.

The wide usage of such metanotations led Guy Steele to call them “the most popular programming language” in a 2017 lecture [39]. However, as Steele argued, the metanotations used for formalizing programming languages have, ironically, not received enough formal examination themselves. One significant issue is the inconsistent use of notations: a survey carried out by Steele and his students identified 28 ways to express substitution, including a pair of notations that mean the opposite thing, but are impossible to distinguish by their appearances [39]. He also mentioned some problems with using overlines to mark repetition.

This frustration with the state of metanotations is what led me to write down “who formalizes the formalization?” as the subtitle of this chapter. It may be true that the inconsistencies highlighted in Steele’s talk do not do hinder PL research. Yet, I still believe we can do better, by bringing a more precise, more consistent metanotation into existence.

1.2 Dreaming of a Computer Science Metanotation

In the aforementioned lecture, Steele suggested that we make Computer Science Metanotation (CSM) “an explicit object of study”: we should formalize CSM, just like we formalize other languages [39]. This is perhaps not a very difficult task: we can incorporate existing formal definitions for CSM elements such as BNF, capture-free substitution, and reduction semantics into a coherent, standardized CSM, so that when researchers try to define their languages, they can use this standardized metanotation.
However, Steele’s outlook has a second component: he suggests that we should also build compilers and other tools for CSM [39], just like we build these tools for other languages. This poses a far greater challenge: many concepts and properties trivial to express in human language are difficult to put into precise, mechanical terms. Name binding and capture-free substitution are perhaps the poster children of this difficulty, spawning a number of wildly different solutions to their modeling, including de Bruijn’s “namefree” expression [14], higher-order abstract syntax (HOAS) [31], and abstract binding trees [23, p. 6-11].

When implementing a “compiler” for CSM, it may be tempting to target proof assistants such as Coq, as they are often used by PL researchers to reason about their languages formally. Unfortunately, the intersection of CSM and proof assistants brings more challenges. In order to express, for example, substitution, the users need to write the entire substitution function by hand in the proof assistant language. The tedious task of writing language-specific boilertplates not only distracts the users from more worthwhile tasks, but also immobilizes them, by tying the abstract syntax of a language to a particular proof assistant implementation, and to one particular representation of variable binding. Tying the CSM to any particular proof assistant may also drive away users who are more familiar with other tools, or wish to approach the same problem with different underlying theories.

There are already many tools that meet many of Steele’s objectives, but fall short of the others. Some, like Ott [38] and Lem [30], handle “compilation” to proof assistants and LaTeX well, but do not provide expressive binding representations. Spoofax [24] and Redex [20, 26] can model complex semantics and reduction rules, but offers limited supports for formal proofs. Perhaps, the lack of an agreed-upon metanotation is exactly the result of the lack of a well-rounded implementation capable of both helping users reason about languages (using formal proofs, for example), and expressing complex language features. The development of a CSM implementation that can both express programming languages definitions with wide ranges of features, and to translate these language definitions to proof assistants may thus be a crucial step on the way towards a standardized CSM.
1.3 **REDEX-PLUS**

Learning from the metanotations in common usage, and past attempts at implementing metanotations, the objective of this project is to bridge the gap between the intuitive modeling languages such as the ones people write in $\LaTeX$, and their implementation as proof assistant code.

In this thesis, I present **REDEX-PLUS**, a CSM implementation. Built upon the Redex library (see section 3.1), **REDEX-PLUS** translates programming language models written in Redex into proof assistant code. The main goal is to allow researchers to define language models, which includes the syntax, reduction rules, and properties of a language, in **REDEX-PLUS**, then use **REDEX-PLUS** to generate proof assistant definitions of their models as well as associated boilerplate, so that they can carry out formal proofs without manually implementing the model in all proof assistants they are using.

**REDEX-PLUS** strives to

- be compatible with Redex, so **REDEX-PLUS** language models are also valid Redex models, allowing Redex’s various useful features, such as reduction evaluation, graphical interfaces, and $\LaTeX$ generation, to remain available to **REDEX-PLUS** users;

- target multiple proof assistants, so a particular model is not tied to one particular proof assistant representation; and to

- support multiple representations of variable bindings in generated code. The differences in binding representations have a significant impact on how formal proofs about a language are written. By providing multiple binding representations, the users may choose the ones most suitable for their particular models and proofs.

**REDEX-PLUS** is open-sourced under the MIT licence. The code repository of **REDEX-PLUS** is publicly available at https://gitlab.com/xujunfeng/redex-plus, and on Zenodo [41].
1.3.1 Structure of this Thesis

In chapter 2, I evaluate some existing metanotations, including Redex, and discuss what we can learn from them. I then introduce the metanotation of REDEX-PLUS, which is a subset of Redex, in chapter 3. After that, I discuss how REDEX-PLUS translates the Redex subset into different proof assistant languages in chapter 4, and the various binding representations it supports in chapter 5. In chapter 6 I discuss some implementation details of REDEX-PLUS. Then, I evaluate REDEX-PLUS in chapter 7, and conclude my thesis with chapter 8.
Chapter 2

Background

Existing Metanotation Implementations

There are a number of metanotation implementations that meet many of CSM’s objectives. Despite the different goals driving the development of these metanotations, in general, they support BNF-like syntax definitions, and various forms of semantic definitions. Some of them also support “compiling” the definition to other languages.

In this chapter, I give a brief description of these metanotations, and evaluate their advantages and drawback. Redex, the metanotation implementation REDEX-PLUS is based upon, is explained in further detail in section 3.1.

To summarize, these are some of the lessons learned from the existing systems:

- For metanotations that can compile to some target languages, the diversity of compilation backends is highly appreciated.

- Modelling safe variable binding in the metanotation is not easy. As Ott (section 2.1) shows, implementing binding in an idiomatic way may require target-specific code.

- Some features not directly related to language modelling are also useful for users: support for IDEs and editors, and generation of \LaTeX\ code are two such examples.
2.1 Ott

Ott is a metalanguage and tool for writing semantic definitions of programming languages [38]. Language specifications in Ott can be “compiled” into a variety of proof assistant languages, including Coq, HOL, and Isabelle, as well as Lem (see section 2.2), and OCaml [37, sec. 1]. Generation of \LaTeX\ representations is also supported. Ott uses a BNF-like syntax for syntax definitions, but allows (and sometimes requires) extra information in the definitions. It also supports relations as “semantic judgements”, defined in a way similar to Gentzen’s derivation trees.

By default, Ott handles binding using a “fully concrete” representation. The user specifies the types (usually strings or natural numbers) used to represent variables in the target proof assistant language, and Ott translates variables in the user-defined model into terms of that type. Ott also generates substitution functions automatically, according to the binding information in syntax definitions. These substitution functions are not capture-avoidant, and there is no “notion of alpha equivalence” [37, sec. 5.3]. As the authors showed in case studies, however, Ott is still able to model many languages, despite the limitations of its binding representation.

Ott comes with experimental support for the locally-nameless (see 5.2) binding representation [37, sec. 11]. An external tool, LNgen (see 2.1.1) can also generate locally-nameless Coq definitions from Ott code.

Among all existing metanotations, Ott fulfills perhaps the largest number of CSM objectives. However, it lacks a built-in notion of reduction or evaluation, so the user must define reductions as relations. While Ott’s syntax is largely similar to functional programming and CSM definitions, the large amount of necessary annotations may clutter the source code. The need to write target-specific annotations also distracts the user from working on models.

2.1.1 LNgen

LNgen is a tool for generating locally nameless (see section 5.2) definitions in Coq from Ott source code [2]. It generates not only definitions of binding operations such as “open” and “close”, but also dozens of useful binding lemmas as well as their proofs. LNgen takes a only subset of Ott as its input. Constructors of variable
arity or non-terminals that are subsets of another non-terminals are not supported. It only accepts binding specifications where “a single metavariable binds in a single nonterminal”.

2.2 Lem

Lem is a “language for engineering reusable large-scale semantic models” [30]. Like Ott, it supports compilation of language models into Coq, HOL, Isabelle, and also OCaml. Lem can also serves as a compilation target of Ott [37, sec. 1] (see section 2.1).

An issue Lem aims to solve is “proof-assistant lock-in”: porting language models between proof assistants is difficult, even though the semantics of the models can be expressed in all these proof assistants. Lem’s metalanguage, whose syntax is “similar to that of a functional programming language”, achieved a “balance” between expressiveness and translatability [30], making definitions written in this metalanguage “reusable”.

Another focus of Lem is human-readability of generate code, in which the whitespaces and comments in the Lem source are preserved, so that the layout of the generated code matches the original. Like Ott, Lem can also target \LaTeX, preserving “typeset layout” in the generate definitions as well. In addition, Lem provides a library of functions, including safely portable ones and some “extra” functions only translatable to some backends.

Compared to Ott, which also targets multiple backends but requires excessive target-specific annotations in the source code, Lem source code is less noisy since annotations are in general not necessary (although there are various target-specific annotations available). This allows users to focus on semantics of the models themselves without paying too much attention to target-specific implementation details. The clarity of the generated code makes them easier to work with. However, Lem’s syntax support is weaker than Ott. It does not provide support for safe bindings in the object language, although users may define a language model with binding specifications in Ott, and compile the Ott code to Lem. \footnote{An example of this is included in Lem’s GitHub repository: \url{https://github.com/rems-project/lem/tree/master/examples/ocaml_light}}.

1
2.3 Spoofax

Spoofax is a “language workbench” for building DSLs [24]. It uses the existing Syntax Definition Formalism language to define language grammars, and Stratego, a framework based on term rewriting [11], to define the semantics of languages. Spoofax is mainly intended for developing new languages and supporting tools. There is little support for writing formal proofs.

A distinct feature of Spoofax is its focus on IDE integration. Spoofax can “automatically derive efficient implementation for various IDE features” from language definitions, and also generate a “stand-alone Eclipse plugin for a language” [24]. This saves the language designer from having to implement IDE support or editor plugins manually.
Chapter 3

REDEX-PLUS

A Computer Science Metanotation

REDEX-PLUS is a computer science metanotation implementation based on Redex, an existing Racket-based metanotation. REDEX-PLUS translates Redex language models into proof assistant code. However, because Redex has many features that are either very difficult or outright impossible to translate to target proof assistant languages, REDEX-PLUS chooses to support only a well-defined translatable subset of Redex. ¹

In this chapter, I first introduce Redex and describe its capabilities, and then I define the translatable subset. I also give an explanation of Redex binding forms, which are important for understanding how REDEX-PLUS handles binding.

3.1 Redex

Redex is a Racket library providing a domain-specific language for defining and working with semantic models [20, 26]. A Redex language model definition usually consists of the following parts:

Language definitions which defines the syntax of terms in a language.

Each language definition contains a list of non-terminal definitions; and each

¹I discuss some possible future extensions to this translatable subset in section 8.2.
non-terminal definition contains a set of patterns. A term is considered to belong a non-terminal if it matches one of the non-terminal’s patterns. Multiple non-terminals can have overlapping sets of patterns, and a term can belong to several non-terminals simultaneously. The language definition also contains a list of binding forms, which specify variable binders in the object language. Object language terms are written in S-expression based concrete syntax: they may have extraneous symbols or brackets. The use of concrete instead of abstract syntax allows greater readability and closer adherence to syntactic norms.

**Metafunctions** (MF), which are functions between terms. Metafunction are not functions in the object language. Instead, they are functions about the terms in the object language, hence the prefix “meta”. A metafunction definition has a list of pattern-matching cases, which are matched against arguments during evaluation. Metafunctions can be recursive. Metafunctions are often used in reduction relation and judgement form definitions.

Redex comes with a built-in substitute metafunction which substitutes a variable with a term. substitute is capture-avoiding.

**Reduction relations** (RR) between terms, representing reduction or evaluation in the object language. Each RR definition contains a list of possible reductions. If a term matches the pattern on the left-hand-side, then it reduces to the term on the right-hand-side.

Given an existing RR, Redex can also automatically construct its closures. The closure can be either a compatible closure, in which all reducible subterms of a term can be reduced; or a context closure, which only allows terms that are in a certain reduction context (defined as a non-terminal) to be reduced.

**Judgement forms** (JF) which are essentially relations over terms. A JF definition contains a set of inference rules from which a true judgement can be

---

2The original spelling is “judgment form”. This thesis uses the “judgement” spelling. Redex-PLUS provides both define-judgement-form and define-judgment-form which are identical except for their names.
derived. A JF can be proven by using the rules to construct a derivation tree either manually or, in some cases, automatically.

With the language model defined, the user may reason about the model in a number of ways: for example, checking a judgement form using `judgment-holds` and automatically finding its derivation trees using `build-derivations`; or calling `apply-reduction-relation` on a term which returns all terms the term may reduce to. Redex also provides a GUI tool for tracing reductions, and procedures for typesetting language models into images or PostScript.

All Redex features involve pattern matching: language definitions use patterns to describe terms’ syntax, while JF, MF and RR definitions work by pattern matching terms against the left-hand-side of rules. Patterns are written using Redex’s powerful `pattern language` in which many more advanced syntactic features in the object language can be expressed succinctly: equality of metavariables is implied by giving them the same “subscript”; and repetition of sub-patterns is denoted by ellipses (…). Redex also allows “unquoting” into the Racket language, so that terms can contain arbitrary Racket expressions, and Racket expressions can be used as `side-conditions` in patterns and RR or JF rules. Racket expressions are useful when computations are required, when expressing arithmetic operations and complicated `side-conditions`.

### 3.1.1 A Basic Example

In this chapter and the rest of the thesis, I use as an example the Polymorphic Lambda Calculus (System F), as described in *Types and Programming Languages* definition [33, p. 343], extended with natural numbers. Figure 3.1 shows an annotated definition of an object language named F, which represents the concrete `syntax` of System F. In figure 3.3, two pieces of System F’s `semantics`, the typing relation and β-reduction, are defined as Redex judgement forms and reduction relations. An additional judgement form and two metafunctions, shown in figure 3.2, are also defined to help express some of the typing rules. Together, the Redex statements in the three figures define the entire `language model` of System F.

To demonstrate how Redex can (visually) help reason with models, figure 3.4 shows the Racket code for generating derivations for the typing judgement “f : N →
; defining a language named "F"
(define-language F
 ; a list of non-terminal specifications:

[t ::= ; the non-terminal "t" can be...
   ; anything that matches the following
   ; patterns:
   (tvar tv) ; a literal "tvar" followed by a
             ; "tv" which is defined later
   nat ; a literal,
        ; representing the natural number type
   (-> t t)
   (forall tv t)]

[e ::=
   (var v) (lam (v : t) e) (app e e) natural
   (tlam (tv) e) (tapp e t)]
[env ::= (cons (v : t) env) (tcons tv env) nil]

; defines "v" and "tv"
; as two distinct types of variables
[v ::= variable]
[tv ::= variable]

#:binding-forms ; a list of binders:

; specifies that lambda-abstraction is a binder,
; and that the lambda argument "v" is bound
; in "e", the function body
(lam (v : t) e #:refers-to v)

(tlam (tv) e #:refers-to tv)
(forall tv t #:refers-to tv)
)

Figure 3.1: The Redex language definition for the F language.
(define-metafunction F
  diff : e v -> boolean
  [[(diff (var v_1) v_1) #f]]
  [[(diff e v_2) #t]])

(define-metafunction F
tdiff : t tv -> boolean
  [[(tdiff (tvar tv_1) tv_1) #f]]
  [[(tdiff t tv_2) #t]])

(define-judgement-form F
  #:mode (In I I O)
  #:contract (In env v t)
  [[(In (cons (v : t) env) v t) "axiom"]]
  [[(In env v_1 t_1)
      (side-condition (tdiff v_1 v_2))
      --------------------------------------------- "cons"
      (In (cons (v_2 : t_2) env) v_1 t_1)]]
  [[(In env v t)
      -------------- "tcons"
      (In (tcons tv env) v t)]]

Figure 3.2: Additional metafunction and judgement form definitions used to define the typing relation in the F language.

\[ \mathbb{N} \vdash f \, 10 : \mathbb{N} \] \(^3\). When executed in an environment that supports a GUI, such as the DrRacket IDE, a window will pop up, showing all possible derivations Redex has found.

3.1.2 Problems with Translating Redex

As the proof assistant targets are less expressive than Redex, accurately translating some Redex features can be difficult if not outright impossible, or require workarounds that may not be worthwhile. Notably, many common idiomatic constructs in Redex’s pattern language are, despite being intuitive for human users to understand, surprisingly difficult to express in proof assistant languages.

When metavariables at different locations in a Redex pattern have the same subscript, the pattern will only match if all such metavariables match the same

\(^3\)"applying \(f\), a function from natural numbers to natural numbers, to 10, produces a natural number."
(define-judgement-form F
#:mode (T I I O)
#:contract (T env e t)

[(In env v t)
  ------------------- "context"
  (T env (var v) t)]

[--------- "natural"
  (T env natural nat)]

[(T (cons (v_a : t_a) env_t) e_b t_b)
  ------------------------------------------- "abstract"
  (T env_t (lam (v_a : t_a) e_b) (-> t_a t_b))]

[(T env e_f (-> t_1 t_2)) (T env e_a t_1)
  --------------------------------------- "apply"
  (T env (app e_f e_a) t_2)]

[(T (tcons tv env) e t)
  ----------------------------------- "Tabstract"
  (T env (tlam (tv) e) (forall tv t))]

[(T env e (forall tv t_a))
  ------------------------------------------- "Tapply"
  (T env (tapp e t_b) (substitute t_a tv t_b)])

(define beta
  (reduction-relation F
  #:domain e
  [--> (app (lam (v : t) e_b) e_a)
      (substitute e_b v e_a) "subst"
      [--> (tapp (tlam (tv) e) t)
          (substitute e tv t) "type-subst"))))

Figure 3.3: The typing judgement and reduction relation for the F language.
(define ds
  (build-derivations
   (T (cons (f : (-> nat nat)) nil)
       (app (var f) 10)
       nat)))
(show-derivations ds)

Figure 3.4: Racket code for generating the list of derivations (ds) and displaying them. Only one derivation is shown as in this example there is only one possible derivation.

term. This notation of equality, which can be used in all Redex statements, does not exist in any of the translation targets. Generally, this means the equality constraints must be made explicit in the target code. For language definitions, which are commonly expressed as data structure definitions, the constraint may take the form of either requiring a proof of equality as an argument to a constructor if the language is dependently-typed, or an additional “well-formed” relation that only holds for terms where the equality is respected. Either implementation makes using the translated language definition more difficult.

Ellipses in patterns, which denote lists of sub-patterns, also pose a challenge. In language definitions, ellipses patterns can be relatively easily translated into lists of data structures representing the sub-patterns. Elsewhere, however, working with the lists in the syntax is difficult, especially when it requires breaking down or
constructing the sub-patterns in lists. For example, a JF rule may state

```
[(P (l expr_1 ...)) (P (l expr_2 ...))
     -------------------------- "zip"
    (P (lp (expr_1 expr_2) ...))]
```

that is, if \( P \) holds for two lists \((l \text{ expr}_1 \ldots)\) and \((l \text{ expr}_2 \ldots)\), then \( P \) also holds for a list of pairs \((lp \text{ (expr}_1 \text{ expr}_2 \ldots))\) formed by “zipping” together the two lists. Representing this rule in the target language would require the use of an ad-hoc “zipping” operation, which, while easy to define in this case, would increase in complexity should the ellipses patterns become more sophisticated. In addition, the two lists are implicitly required to be of the same length, adding an additional constraint that must be caught and preserved by the translation process.

Combining implicit metavariable equality and ellipses patterns, it is possible to express the recurrence of sub-patterns in a list. For instance, the pattern \((\text{expr}_1 \text{ expr}_2 \ldots \text{ expr}_1 \text{ expr}_3 \ldots)\) matches a list in which the first sub-pattern \((\text{expr}_1)\) appears again later. Matching such patterns in the target language may require traversing the list.

As mentioned before, multiple Redex non-terminals can have overlapping sets of patterns. This makes it possible to, for instance, in a manner similar to the informal notations commonly seen in PL literature, define a “value” non-terminal whose pattern set is a strict subset of the “term” non-terminal’s, so that every “value” (which usually means “a term that cannot be further reduced”) is also syntactically a term. This works in Redex where object language terms do not carry “types” intrinsically. Unfortunately, in all backends REDEX-PLUS targets, object language terms are idiomatically represented by typed data structures, thus each term can only be of one type\(^4\).

Apart from the hard-to-translate patterns, another difficulty is translating arbitrary Racket expressions that can appear in terms in the object language or as side-conditions. Because Racket is a full-fledged language with a large standard library that is never formalized, it is difficult to faithfully translate Racket

---

\(^4\)In some targets this can be worked around using subtyping possibly in conjunction with type classes, but this is not idiomatic.
expressions into proof assistant targets.

3.2 The Supported Redex Subset

REDEX-PLUS supports a subset of Redex with most of Redex’s hard-to-translate features removed. This translatable subset is sufficiently well-defined to allow accurate translation into different proof assistant targets, and at the same time, sufficiently expressive to represent a wide range of language models. Notice that the aforementioned Redex model of System F (shown in figure 3.1) is defined using only features from the supported subset.

REDEX-PLUS supports all the four main Redex statements for defining language models, described at the beginning of section 3.1. Macros for reasoning about models, like judgment-holds, are not supported, as to translate them amounts to generating the code that carries out the reasoning tasks in the target proof assistant. Because the purpose of REDEX-PLUS is translatable language models, rather than doing the proofs, these macros are beyond the scope of this project.

Some possible extension to the translatable subset are discussed in section 8.2.

Patterns and Terms

REDEX-PLUS uses a small, strict subset of the original pattern language as shown in figure 3.5. Most of the Redex pattern language, including ellipses patterns (pattern-sequence as known in the Redex pattern language), side-conditions, all patterns related to holes or closures, strings, and Racket numbers\(^5\), are removed. natural and integer, the two basic numerical types supported across a wide range of targets, are however still supported.

---

\(^5\)Racket implements the full Scheme numerical tower including arbitrarily large and complex numbers, which are not well-supported across targets. However, not supporting the full numerical tower has only marginal impact on language modeling.
pattern = natural  
    | integer  
    | variable  
    | (variable-prefix id)  
    | symbol  
    | other-literal

Figure 3.5: Pattern language supported by REDEX-PLUS.

REDEX-PLUS still supports using unquote to include Racket expressions in terms. However, the Racket expressions can only contain four basic arithmetic operations: addition, subtraction, multiplication, and division. Division is supported only for the SMT-LIB target, as other targets do not support non-total operations.

Language Definitions

REDEX-PLUS imposes some additional restrictions on language definitions. The pattern set for each non-terminal must be disjoint, in other words, no pattern can belong to multiple non-terminals at the same time. In non-terminal definitions, REDEX-PLUS requires each pattern that is not a binder specified by a binding form to contain no more than one potentially-bound variable. In addition, in binding forms, only one variable may be bound at one time. A pattern with one variable as its only argument is called a variable constructor. In the generated target code, when substituting a variable for a term, the entire term constructed by the variable constructor (instead of just the variable inside the term, as is the case with Redex) will be replaced.

The statement define-extended-language which creates a new definition by “extending” an existing one with additional non-terminal patterns, is not supported. This is because in Redex a term can belong to both the original and the extended language at the same time, but the target languages, it is not possible to “extend” a data structure definition defining the syntax of a language; and if the extended language is defined as a new language, then a term cannot be in both languages at the same time, as the “same” term will be represented differently in the original and the extended language.
Notice that many of REDEX-PLUS's restrictions on bindings, overlapping non-terminals, as well as the previously mentioned ones on ellipses in patterns, are similar to the restrictions imposed by LNgen on Ott (see 2.1.1).

**Judgement Forms**

Redex allows use of ellipses in judgement form definitions in a way similar to ellipses in patterns. REDEX-PLUS does not support these ellipses. However, although while clauses and side-conditions are not supported in either patterns or as a part of MF and RR cases, they are allowed in JF cases.

**Metafunctions**

The range of a Metafunction’s signature must be either a non-terminal in the object language, or boolean. Disjunctive (or) ranges are not supported. The signature may not contain pre- or post-conditions. MF cases cannot have extra conditions denoted using where, side-condition, or judgement-holds.

Importantly, MFs are required to be total. Redex allows non-total MF definitions, only raising errors during runtime when the MF is applied to undefined arguments. However, most of the target languages forbid non-total function definitions outright. While REDEX-PLUS does not come with a totality checker for MFs, non-total MF definitions will almost certainly be translated into illegal definitions in the targets where functions must be total.

**Reduction Relations and Closures**

Only the very basics of reduction relation definitions are supported. Like MF cases, RR cases are not allowed to have extra conditions denoted using where, side-condition, or judgement-holds. They cannot use fresh to generate fresh variables either. In terms of syntax of the RR statement, this means all of red-extras except rule-name are not supported. Shortcuts in definitions are not supported either.

The macro compatible-closure is also supported, but context-closure is not supported. This is because a context-closure requires a reduction context. Redex uses special non-terminal definitions as reduction contexts, and these non-
terminal definitions are difficult to translate, due to aforementioned problems with overlapping non-terminal patterns. Some possible workarounds are discussed in section 8.2.
Chapter 4

Translating Language Models, and Generating Code

REDEX-PLUS can translate a Redex language model into one of the target proof assistant languages, so the properties of a language model can be proven or verified using the proof assistants. In this chapter, I will describe how aspects of the Redex language models are represented in the target languages, with an emphasis on various ways of representing variable bindings.

Throughout this chapter, I will use the following two language models as examples:

The **Tree language** defines Peano numbers and a binary tree containing Peano numbers. As a simple language with no variable bindings, it will be used in the description of the basic translation procedure.

The **F language** introduced in section 3.1 and defined in figure 3.1. The F lan-

```scheme
(define-language Tree
    [n ::= z (s n)]
    [bt ::= (leaf n) (bt_l branch bt_r)]
)
```

**Figure 4.1**: The Tree language model.
guage represents Polymorphic Lambda Calculus or System F, with support for natural numbers. It involves almost all of REDEX-PLUS’s features, including, most importantly, multiple variable types and their binding rules.

4.1 Language Models to Intermediate Representation

REDEX-PLUS translates a language model in two steps. First, it translates the model into Poultry, a target-independent intermediate representation (IR). And then, it generates code in the target proof assistant languages from the IR. Compared to translating Redex language models into target languages directly, using an IR allows much of the code to be shared between the translation processes for different targets, while also making the development of new translation targets faster. I will describe the implementation in more detail in chapter 6.

4.1.1 Poultry: The Intermediate Representation

The intermediate representation Poultry\(^1\) provides the following constructs:

**for-language** defines an environment in which Poultry statements refer to a certain Redex language. Since all Redex definitions supported by REDEX-PLUS must declare the Redex language they refer to, all Poultry statements must be enclosed in for-language environments as well.

For the sake of simplicity the for-language environment is omitted from code examples. This would not cause confusion as all the code in an example (other than the one in figure 4.2) belongs to the same Redex language.

**define-type** defines a new type in a language, and the constructors that construct terms of this type. The constructors of a type can be recursive, but it must be possible to construct a finite term for every type. Some recursive constructors can construct a term that binds the variables in its subterms, and some constructors may be bound variables.

One can use define-types to define multiple types at once. Types defined together in this way can have mutually recursive constructors.

---

\(^1\)Named so because Coq and Agda, two important translations targets, are avian.
Function and proposition definitions must only use types already defined.

**define-function** defines a function from a term in the language to either another term in the language, or a boolean value. It supports pattern matching which can be used to deconstruct terms, conditional expressions, and some basic arithmetic operations.

A function definition must be total. Otherwise, the function definition is malformed, and the generated proof assistant representation can be erroneous.

**define-function** can also be used to define nullary “functions”. This feature is used in parts of the translation processes to define constants.

**define-proposition** defines a first-order predicate over terms in the language, as well as the conditions under which the predicate is true, which can be recursive. The conditions can refer to functions previously defined using **define-function**.

These three Poultry definitions are sufficiently expressive to represent all major features of Redex language models, namely, language definitions, judgement
forms, metafunctions, and reduction relations and their closures. At the same time, they are basic, generic concepts not tied to any particular representation, supported by proof assistants and other tools built upon different background theories ranging from dependent types to many-sorted logics, which makes the final translation step from Poultry to target code straightforward and relatively easy to implement.

As Poultry definitions are intended to be used only in the automated translation process rather than being hand-written, they are not accessible to REDEX-PLUS users.

I will describe in further detail how the definitions translate into target code in section 4.2.

### 4.1.2 From Redex To Poultry

#### Language Definitions

A `define-language` statement, which defines a new Redex language with one or more non-terminals, is translated into a single `define-types` definition in Poultry, in which each type definition corresponds to one non-terminal. Here a single `define-types` statement is used instead of multiple `define-type` statements, because the latter does not support mutually recursive non-terminal definitions.

Each Redex non-terminal definition contains a list of patterns of valid terms. REDEX-PLUS represents each of these patterns as a constructor. The name of each constructor is derived from the literals in the pattern or, if no literal exist, the names of the non-terminals in the pattern. The Redex patterns represent the concrete syntax of the language. They may contain parentheses and multiple literal symbols, which improves readability when designing and presenting language models, but they are poorly supported by REDEX-PLUS’s targets. REDEX-PLUS process simplifies patterns by taking out the parentheses, and removes all literals, leaving only other non-terminals as the types of constructor arguments, effectively transforming the concrete syntax definitions into an abstract syntax. All terms and patterns in other Redex definitions are simplified using the same process.

When a non-terminal is defined with more than one names in Redex, the first name will be considered canonical and be used in the generated IR, and the other
Figure 4.3: Defining a binary tree type with define-types.

```
(define-types (bt n)
  (((leaf (n))
    (branch (bt bt)))
  ((z ()) (s (n)))))
```

Figure 4.4: The Redex metafunction that mirrors a binary tree.

```
(define-metafunction Tree
  mirror : bt -> bt
  [[(mirror (leaf n)) (leaf n)]
   [[(mirror (bt_1 branch bt_2))
     ((mirror bt_2) branch (mirror bt_1))])
```

Figure 4.3 shows how the Poultry type definition corresponding to the Redex Tree language definition. Notice that the branch constructor is placed before its arguments like all other constructors, rather than between two sub-trees like in the original language model.

**Metafunctions**

A define-metafunction definition is translated into a define-function statement. The body of the metafunction, which is a list of patterns, is translated into a single pattern-#:match expression as the body of the definition. The type signature of the original statement is preserved in the generated definition, and some additional properties are passed into the definition, as they will later be used for generating proof assistant code.

Figure 4.4 and 4.5 shows how a simple binary tree mirroring function is translated.

In Redex, if a pattern variable appears more than once in a pattern, then the names, or aliases, will be replaced by the first canonical name.

Figure 4.3 shows how the Poultry type definition corresponding to the Redex Tree language definition. Notice that the branch constructor is placed before its arguments like all other constructors, rather than between two sub-trees like in the original language model.
(define-function mirror
  #t ; should the language name be prefixed?
  #f ; does the language have binding?
  #f ; is the function recursive?
  (t1); list of argument names
  (bt); list of argument types

; the function body
(#:match (t1)
  (((leaf n))
   (leaf n))
  (((branch bt_1 bt_2))
   (branch (mirror bt_2) (mirror bt_1))))

bt ; the return type
)

Figure 4.5: The generated IR of the statement in figure 4.4, with all arguments annotated. All user-defined metafunctions will have the language name prefixed to the function name.

pattern will only be matched if all occurrences of the pattern variable matches the same term. This is not directly supported by Poultry or any of the targets, so Redex-Plus must make the implied equality of patterns explicit during translation.

Given a metafunction case whose pattern contains multiple occurrences of the same pattern variable, Redex-Plus translates it by first pattern matching normally, and then replacing the result of the pattern match with an #:if expression with the equality between patterns as its condition. The true branch of the #:if expression, which is returned when the equality condition is satisfied, is the term to be returned should this case match; and the false branch is another pattern match with rest of the patterns in the list translated similarly.

Sometimes, the generated function body contains unnecessary #:match expressions. Redex-Plus is able to detect and simplify these expressions before generating target code, to keep the language definition as simple as possible.
(define-metafunction Tree
    same-tree : bt bt -> boolean
    [(same-tree bt_1 bt_1) #t]
    [(same-tree bt_1 bt_2) #f]
)

Figure 4.6: The Redex metafunction that compares two binary trees. The first case, which returns true, is only matched when both arguments are syntactically the same. If they are not, then the function falls back to the second case and returns false.

An example of a function with multiple occurrences of the same pattern variable and the corresponding Poultry definition is shown in 4.4 and 4.5. Notice that in the generated function body, there are two useless #:match expressions. Figure 4.8 shows that, after simplification, both #:match expressions are removed and the function body is reduced to one simple conditional expression.

Judgement Forms

Each define-judgement-form statement in the language model is translated into one single Poultry define-proposition definition. Each rule for constructing a judgement in the Redex statement corresponds directly to a rule in the IR. Figure 4.9 and 4.10 shows a definition representing the “less-than” relation between two Peano numbers in the Tree language, and the resultant IR.

For simple judgement form rules such as the ones in figure 4.9, the translation from Redex rules into Poultry is straightforward, requiring little processing apart from term simplification. Some language models make use of side conditions and where clauses, which require additional processing:

- Each side-condition, which evaluates to a term-level \(^2\) boolean value, is wrapped up in a #:is-true? term, elevating it to the proposition level in the target language.

\(^2\)In dependently-typed proof assistants, the boolean data type is the term-level boolean, and propositions are proposition-level. In SMT-LIB, both term- and proposition-level booleans are represented by the native boolean type.
(define-function
same-tree
#t ; should the language name be prefixed?
#f ; does the language have binding?
#t ; is the function recursive?
t4 t5 ; list of argument names
(bt bt) ; list of argument types

; the function body
#:match
t4 t5
((a0 a1)
#:if
#:bool-and (#:bool-equal? bt a0 a1)) ; returns true if the trees are equal
(true3844) ; returns false otherwise
#:match (t4 t5) ((bt_1 bt_2) (false3845)))

#:bool ; the return type
)

Figure 4.7: The IR of the same-tree function definition. Notice the #:if expression inside the top level #:match, in which the two arguments are compared.

; the simplified function body of same-tree
#:if (#:bool-and (#:bool-equal? bt t2 t3)
(true3818) (false3819))

Figure 4.8: The function body of same-tree after simplification.
(define-judgement-form Tree
  #:contract (Le n n)
  [------------ "Le-zero"
    (Le z (s n))]

  [(Le n_1 n_2)
    ------------ "Le-succ"
    (Le (s n_1) (s n_2))]]

**Figure 4.9:** The Redex judgement representing the “less-than” relation between Peano numbers.

(define-proposition
  Le (n n)
  ("Le-zero"
    #:implicit (n)
    #:result-type (Le (#:term (z)) (#:term (s n)))
    ()
  )

("Le-succ"
  #:implicit (n_1 n_2)
  #:result-type (Le (#:term (s n_1))
                (#:term (s n_2)))
  (((Le (#:term n_1) (#:term n_2))))

**Figure 4.10:** The generated IR of the statement in figure 4.9. The first two arguments of **define-proposition** declares the name and signature, in terms of type names, of the judgement. A list of rules follows, in which each rule contains a list of (implicitly) bound variables used in the rule, the judgement constructed by the rule, and a potentially empty (in the case of axioms) list of premises. The #:term keyword helps the later translation process distinguish judgement forms from terms.
define beta
(reduction-relation
F
#:domain e
[--> (app (lam (v : t) e_b) e_a)
    (substitute e_b v e_a) "subst"
][--> (tapp (tlam (tv) e) t)
    (substitute e tv t) "type-subst"]))

Figure 4.11: The Redex definition of $\beta$-reduction in System F.

- Each where clause, which compares two terms of the same type, is turned into a proposition-level comparison between the two terms.

Reduction Relations as Functions

Reduction relations can be considered as functions that take terms and return a potentially empty list of possible reduction results. This representation of reduction relations is closer to Redex’s way of representing reduction relations: as reduction steps that can be evaluated (see section 3.1).

When treating reduction relations as functions, each reduction case is translated into an individual function definition, named after the reduction case. If the argument to the function matches the left-hand-side of the reduction case, the function returns a list with the right-hand-side as its sole element. Otherwise, it returns an empty list. The reduction relation itself is then represented by the list of reduction case functions.

The final generated proof assistant code may come with a helper function that applies the list of functions to a term, returning a list of all possible reduction outcomes. The generated code may also provide a relational definition of the reduction relation, by defining the relation as “$t$ reduces to $t'$ if $t'$ is in the list of possible reduction outcomes of $t$”.

Figure 4.11 shows the Redex definition of $\beta$-reduction for System F; figure 4.12 shows the function definition corresponding to the subst case.
(define-function beta-subst
  #f ; language name should not be prefixed
  #t ; the language has binding
  #f ; the function is not recursive
  (t) ; list of argument names

  ; the argument should be a well-bound expression
  ((#:well-bound e))

  ; the function body
  (#:match (t)
    (((app (lam t e_b) e_a))
      ; #:substitute is a special function definition
      (#:list (#:substitute e_b v e_a)))

    ; the #:any pattern matches all terms
    ((#:any) (#:list))

  ; returns a list of well-bound expressions
  (#:list-of (#:well-bound e)))

Figure 4.12: The function definition corresponding to the subst reduction case. The argument type (#:well-bound e) means an e that has a type-level guarantee of its well-boundedness. The well-boundedness requirement is only relevant when using binding representations that can express well-boundedness, such as shifted names. When using other binding representations, the requirement is ignored.

Reduction Relations as Relations
Alternatively, reduction relations can be understood as relations between terms and their possible reduction outcomes. This results in a simpler representation that is easier to work with in proofs, but cannot be evaluated like functions.

Similar to Redex judgement forms, each reduction relation is translated into a single define-proposition definition. Each reduction rule corresponds to one rule in the IR, with no premises, and “the left-hand-side of the original rule reduces to the right-hand-side of the original rule” as the conclusion. Figure 4.13
(define-proposition
  beta (e e)
  ((beta-subst
    #:implicit (t e_b e_a)
    #:result-type
    (beta (#:term (app (lam t e_b) e_a))
      #:term (#:substitute e_b v e_a)))
    ()))
  (beta-type-subst
    #:implicit (e t)
    #:result-type
    (beta (#:term (tapp (tlam e) t))
      #:term (#:substitute e tv t)))
  ())))

Figure 4.13: $\beta$-reduction of System F represented as a proposition.

shows how $\beta$-reduction for System F, defined in Redex in figure 4.11, can be thus translated into a proposition definition.

**Compatible Closures as Relations**

Similarly, compatible closures can also be represented as relations, and translated into `define-proposition` definitions. Represented as a relation, the compatible closure of an existing reduction relation on a non-terminal $e$ contains all rules in the `define-proposition` statement representing the original reduction relation as well as new rules covering reduction of sub-terms. Each new rule covers one recursive occurrence of $e$ as an argument to a constructor of $e$, with “$e\text{\_from}$ reduces to $e\text{\_to}$” as its sole premise, and “$(c \ a_1 \ldots \ e\text{\_from} \ldots \ a_n)$ reduces to $(c \ a_1 \ldots \ e\text{\_to} \ldots \ a_n)$” as its conclusion, where $c$ is a constructor of $e$. Both $e\text{\_from}$ and $e\text{\_to}$, and the other arguments in the constructors are universally quantified.

Figure 4.14 shows how the compatible closure of $\beta$-reduction for System F, defined in 4.11 and translated into a relation in 4.13, is represented as a relation definition. Note that there are two rules covering the reduction of sub-terms in the `app` (function application) constructor, as `app` takes two `es` as arguments.
Figure 4.14: Compatible closure of $\beta$-reduction of System F represented as a proposition; abridged.

(define-proposition beta
  (e e)
  ; The original rules
  ((beta-single-subst-cc
    #:implicit (t e_b e_a)
    #:result-type
    (beta (#:term (app (lam t e_b) e_a))
      #:term (#:substitute e_b v e_a))
    )))
  ; beta-single-type-subst-cc ...

  ; the new rules
  ("beta-cc-lam"
    #:implicit (e_from e_to t_i1)
    #:result-type
    (beta (#:term (lam t_i1 e_from))
      #:term (lam t_i1 e_to))
    ((beta e_from e_to)))

  ; two rules for the "app" constructor
  ; as there are two es in it
  ("beta-cc-app"
    #:implicit (e_from e_to e_i1)
    #:result-type
    (beta (#:term (app e_from e_i1))
      #:term (app e_to e_i1))
    ((beta e_from e_to)))
  ("beta-cc-app-2"
    #:implicit (e_from e_to e_i0)
    #:result-type
    (beta (#:term (app e_i0 e_from))
      #:term (app e_i0 e_to))
    ((beta e_from e_to)))

  ; other constructors follow
  )
4.2 Generating Code

After the first translation step during which a Redex model is translated to Poultry, the Poultry IR is then translated into target proof assistant code in the second step. At a high level, the three Poultry constructs are translated as follows:

- `define-type` translates to a data definition in the target.
- `define-function` translates to a function definition in the target language.
- `define-proposition` translates to something that can be proven to be true by the target.

The above describes the intent of each Poultry construct that should be preserved in the translated proof assistant code, rather than concrete ways in which the translation is carried out, because REDEX-PLUS targets proof assistants with a wide range of background theories, making it difficult to strictly specify what each Poultry construct corresponds to in the target language. For example, propositions can be expressed idiomatically as type definitions in dependently-typed languages, but not in SMT solvers.

When translating a Redex source file, on top of translating each Poultry definition, a predefined prelude is inserted at the beginning of the output target code. The content of the prelude differs between different proof assistant targets and binding representations, but, in general, the prelude imports libraries, standard library data types, and defines data structures and helper functions that might be used by the generated code.

To help distinguish identifiers from different Redex languages from one another, the names of the Redex language they belong to are prefixed to the names of types, constructors, functions, et cetera in the target code. In addition, if an identifier contains characters not allowed in identifier names in the target language, the characters are replaced during code generation.

In the rest of this section, I explain how Poultry constructs are translated into different proof assistant targets, as well as the differences in the generated code arising from the different underlying theories used by the targets.
4.2.1 Dependently Typed Proof Assistants

The translation processes from Poultry IR into Coq, and Agda, the two dependently-typed targets, are largely similar, despite the numerous differences in the targets’ syntax, theorem-proving style, and underlying theories. In this section I describe how Poultry constructs are translated into dependently-typed target languages, before explaining some target-specific quirks and treatments.

Type Definitions

Poultry type definitions are translated to idiomatic data structure definitions in dependently-typed targets. Native types in constructor arguments are replaced by types supported by the target’s standard library, and variables are replaced by target- and binding-representation-specific data structures containing necessary binding information. There is little complexity in the translation process as the correspondence between the Poultry define-type definition and target code is almost one-to-one.

For each type definition, REDEX-PLUS also generates the boolean equality (eqb \(^3\)) definition for terms of the type. For a type \(t\), its boolean equality definition is a function with the signature eqb : \(t \rightarrow t \rightarrow \text{Boolean}\), where Boolean is the standard boolean type in the target language, returning true if the two terms are the same and false if otherwise. The boolean equality definition is used when translating metafunction definitions with implied equality conditions, previously discussed in section 4.1.2.

If the type definition contains bound variables, REDEX-PLUS also generates helper functions for binding variables, as well as substituting terms. This is discussed in detail in chapter 5.

Function Definitions

Poultry function definitions are translated into function definitions relatively straightforwardly. However, there are major differences in how pattern matching, which is necessary for representing Redex metafunctions, is handled in different targets, as discussed later in this section.

\(^3\)Borrowed from Coq.
**Proposition Definitions**

In dependently typed languages, propositions are idiomatically defined as data structure types, where the constructors can be used to construct proofs of the proposition. This closely corresponds to the Poultry `define-proposition` statement which contains a list of rules for proving the proposition. The translation is therefore straightforward: The judgement form itself is translated into a type definition, parameterized by the non-terminals in the signature of the proposition definition; and each rule is translated into a constructor, with the name of the rule as the name of the constructor, the list of `#:implicit`ly bound variables as the list of arguments taken by the constructor, and the `#:result-type`, i.e. the judgement constructed by the rule, as the return type of the constructor.

**Targeting Specific Dependently-Typed Targets**

**Coq** is a proof assistant with the calculus of inductive constructions, a dependent type system, as its underlying theory [9]. Due to Coq’s popularity in the wider ecosystem, Coq is the most important translation target, with the most features supported.

When multiple types are defined at once using `define-types`, the type definitions are joined together using `with` so that they may be mutually dependent.

Proofs in Coq are idiomatically written as sequences of proof steps called *tactics*. Tactics can be combined and reused to simplify proofs, and to allow some degree of automation [17]. Whenever a proposition named \( x \) is

---

**Figure 4.15:** The Coq definition of a binary tree generated by REDEX-PLUS
translated into Coq, a new tactic named autoderive_x is generated. As its name suggests, the tactic automatically derives a proof, by making repeated attempts at applying the proposition’s rules to the goal. It succeeds if it makes any progress, and fails if it does not. REDEX-PLUS also generates an autoderive (without suffix) tactic, which tries to apply all previously defined proposition-specific autoderive tactics, saving the user from having to type out the full name of the autoderive tactic they wish to use.

The autoderive tactic is generated for all propositions, including not only the ones representing judgement forms, but also the ones representing reduction relations and metafunctions.

Figure 4.16 shows the “less than” relation between natural numbers, as defined in figure 4.9, translated into Coq, along with the generated autoderive definition. Figure 4.17 shows how autoderive can be used to write succinct proofs. Note that the user needs to perform induction, and to introduce new assumptions or variables manually, as autoderive makes no attempt at these actions.

**Agda** is a dependently typed language based on Martin-Löf type theory that is also a proof assistant [10]. Figure 4.18 shows the type definitions for Peano numbers and binary tree, as well as the “less than” relation between numbers.

Functions are translated into declarations followed by definitions, where the definitions are translated differently depending what is inside the function body. In simple cases, where the body has only one top-level pattern matching expression, the entire function body is translated into one anonymous lambda abstraction expression. In more complicated cases, where there are pattern matches inside the function body, REDEX-PLUS extracts pattern matches into helper functions, places the helper functions in a where clause, and replaces their occurrences with calls to the helpers.

The extraction of inner pattern matches is necessary because Agda does not have a pattern-matching expression, so pattern matching has to be done in function definitions, and while Agda does support pattern matching in lambda expressions, it has problem type checking the expressions if they are inlined...
Inductive Tree_Le : Tree_n -> Tree_n -> Prop :=
| TreeProp_Le_zero {n} : (Tree_Le (Tree_z ) (Tree_s n))
| TreeProp_Le_succ {n__1 n__2} : (Tree_Le n__1 n__2) ->
   (Tree_Le (Tree_s n__1) (Tree_s n__2)).

Ltac autoderive_Tree_Le :=
  (apply TreeProp_Le_zero; autoderive_Tree_Le) +
  (apply TreeProp_Le_succ; autoderive_Tree_Le) +
  (progress auto; autoderive_Tree_Le) +
  (progress compute; autoderive_Tree_Le).

Ltac autoderive :=
  first [autoderive_Tree_Le].

Figure 4.16: The “less than” relation defined in figure 4.9 translated into Coq,
and the associated autoderive definitions.

Theorem N_Less_Than_N_Plus_Two :
  forall n, Tree_Le n (Tree_s (Tree_s n)).
Proof.
  intro n.
  induction n.
  all: autoderive.

(* The alternative proof body:
  - apply TreeProp_Le_zero.
  - apply TreeProp_Le_succ.
  auto. *)
Qed.

Figure 4.17: Proving that a natural number is less than itself plus two by
induction on natural numbers. Both induction cases can be trivially
solved by autoderive.
\begin{verbatim}
data Tree-bt : Set
data Tree-n : Set
data Tree-bt where
Tree-leaf : Tree-n → Tree-bt
Tree-branch : Tree-bt → Tree-bt → Tree-bt
data Tree-n where
Tree-z : Tree-n
Tree-s : Tree-n → Tree-n
data Tree-Le : Tree-n → Tree-n → Set
data Tree-Le where
TreeProp-Le-zero : { n : Tree-n } →
(Tree-Le (Tree-z) (Tree-s n))
TreeProp-Le-succ : { n:1 : Tree-n } →
{ n:2 : Tree-n } →
(Tree-Le n:1 n:2) →
(Tree-Le (Tree-s n:1) (Tree-s n:2))
\end{verbatim}

\textbf{Figure 4.18:} The Agda definition of a binary tree, as well as the “less than” relation between Peano numbers.

in the main function body, so these helper functions must be explicitly declared and defined in a \texttt{where} clause.

\subsection*{4.2.2 Beluga}

Beluga is a programming language for reasoning about formal systems. In Beluga, bindings are represented using higher-order abstract syntax (HOAS), in which bindings in the object language being modelled are represented by bindings in its Beluga representation [32].

Since the idiomatic Beluga representations of propositions and functions differ too much from Redex and other targets, currently \textsc{REDEX-PLUS} can only translate
Figure 4.19: System F represented in Beluga. The natural number type nat is defined in the prelude.

Type definitions into Beluga.

Translating types and propositions into Beluga is straightforward and largely similar to corresponding translations into dependently-typed targets. Both type and proposition definitions are translated into type declarations, followed by constructor declarations that corresponds to the type constructors, or the proposition rules. What is unique about the Beluga target is that it only supports HOAS binding representations, which is discussed further in section 5.4.

4.2.3 SMT-LIB

Satisfiability modulo theories (SMT) solving is about determining the satisfiability of logic formulae with regard to certain logic theories [6] such as arithmetics or quantifiers [15]. As an extension of traditional satisfiability (SAT) solvers with additional theories, SMT solvers can be implemented by “interfacing” an SAT solver with solvers for the theories involved [16]. Like SAT solvers, SMT solvers can be used to find models that satisfy certain properties (for example, that the program can reach an illegal state) and prove the correctness of a formula by showing that its negation is not satisfiable. Use cases of SMT solvers include model checking [1] [13], and optimization [29].
Compared to other translation targets, SMT solvers are unique in their capability for:

- Reasoning about arithmetic properties efficiently;
- Coming up with models that satisfy certain properties; and
- Operating automatically, requiring little user input for finding proofs.

Many modern SMT solvers, such as Z3 [15] and CVC5 [5] accept inputs in the 2.6 version of the SMT-LIB language [7]. The SMT-LIB 2.6 standard defines an s-expression-based language which is used to both define the model to be checked and to instruct the SMT solver to check the model. SMT-LIB 2.6 supports not only various arithmetic theories, but also features useful for modeling computer programs, such as recursive data types and functions. REDEX-PLUS supports translating type and proposition definitions into SMT-LIB.

**Translation into SMT-LIB**

Poultry types are translated into SMT-LIB datatypes using the `declare-datatypes` commands. Like with Coq, when multiple types are defined together in a single `define-types` statement, they are declared together so the types can be mutually recursive. Unlike in other targets, constructor arguments in SMT-LIB are named, so REDEX-PLUS automatically generate unique names for them.

Proposition definitions are translated into *uninterpreted* functions. First, a proposition is declared as a function from its arguments to a boolean using the `declare-fun` statement. Then, the rules in the proposition definition are represented as assertions, stipulating that the function returns true for certain inputs if certain premises are true. Uninterpreted functions are used because due to difference between SMT-LIB’s function definition syntax and that of Redex, as well as SMT-LIB’s lack of a pattern-matching expression, it is difficult to represent the Redex definitions as concrete SMT-LIB functions.
Chapter 5

Variable Binding

Representation & Implementation

Thus was he given his name by one very wise in the uses of power.

(Le Guin, *A Wizard of Earthsea* [27])

Building upon Redex’s binding forms discussed in section 3.1, REDEX-PLUS is capable of expressing variable bindings and translating them in a variety of ways. In this chapter, I describe the binding representations REDEX-PLUS supports, how substitution and other binding-related operations are implemented, and how each of these representations look like in generated target code.

For this chapter, I use the F language defined in figure 3.1 as the example. The F language defines System F [33, p. 343] with two variable types (term variables v, and type variables tv). To my knowledge, so far REDEX-PLUS is the only metanotation capable of generating binding implementations that 1) involve multiple variable types, 2) are target agnostic, requiring no target-specific code in the language model ¹, and 3) target multiple proof assistant languages.

Prequel: De Bruijn Indexing

De Bruijn indexing [14] is one of the earlier binding representations. Because language models using de Bruijn indexing often need to formulate judgement form

¹Contrast LNGen (see 2.1.1).
definitions in specific ways incompatible with other binding representations, it is often not possible to use de Bruijn indexing as a drop-in replacement for other binding representations. In contrast, the named, locally nameless, and shifted names representations, discussed later in this chapter, are drop-in replacements for each other. For this reason, REDEX-PLUS does not support de Bruijn indexing. But because de Bruijn indexing forms the basis upon which some other representations are built, I introduce this approach here for the sake of clarity.

With de Bruijn indexing, all variables are represented as natural numbers, and names are erased from variable references and binders. The index representing a variable is the number of binders in the syntax tree between a variable and its own binder. If the index is greater than the number of bindings in existence, then the variable is free, and its name is kept in a “free variable list” which is not a part of the syntax tree.

Two terms written using de Bruijn indexing are \( \alpha \)-equivalent if and only if they are syntactically identical. This greatly simplifies checking for \( \alpha \)-equivalence. However, it is difficult for human users to read and understand terms written using de Bruijn indexing. The locally nameless and shifted names representations, both based on de Bruijn indexing, both try to improve human usability while preserving the \( \alpha \)-equivalence property.

### 5.1 Named

A simple way to deal with variables is to represent them as their textual names. Substitution would be defined as a function that takes the name of the variable to be substituted for, the term to be substituted with, the term in which the substitution happens, and returns the term with all free occurrences of the variable replaced. It is crucial that substitution changes only the free occurrences of a variable but not bound ones, as they are supposed to denote different things despite possibly sharing a name.

When translating language models using this named approach, all variables in patterns are represented by native string types in the target. This includes variables in variable constructors, variables in binding forms, and variables with no binding involved. Figure 5.1 shows the System F definition translated into Coq us-
\textbf{Inductive} \(F_e : \text{Type} :=\)
\begin{itemize}
  \item \(F_{\text{var}} : \text{string} \rightarrow F_e\) (* var. constructor for \(F_e\) *)
  \item \(F_{\text{app}} : F_e \rightarrow F_e \rightarrow F_e\)
  \item \(F_{\text{e\_natural}} : \text{nat} \rightarrow F_e\)
  \item \(F_{\text{tapp}} : F_e \rightarrow F_t \rightarrow F_e\)
  \item \(F_{\text{tlam}} : \text{string} \rightarrow F_e \rightarrow F_e\) (* Binds \(F_{\text{tvar}}\) *)
  \item \(F_{\text{lam}} : \text{string} \rightarrow F_t \rightarrow F_e \rightarrow F_e\) (* Binds \(F_{\text{var}}\) *)
\end{itemize}
\textbf{with}\n\begin{itemize}
  \item \(F_t : \text{Type} :=\)
  \begin{itemize}
    \item \(F_{\text{tvar}} : \text{string} \rightarrow F_t\) (* var. constructor for \(F_t\) *)
    \item \(F_{\text{nat}} : F_t\)
    \item \(F_{\text{___arrow__}} : F_t \rightarrow F_t \rightarrow F_t\) (* binds \(F_{\text{tvar}}\) *)
    \item \(F_{\text{forall}} : \text{string} \rightarrow F_t \rightarrow F_t\)
  \end{itemize}
\end{itemize}
\textbf{with}\n\begin{itemize}
  \item \(F_{\text{env}} : \text{Type} :=\)
  \begin{itemize}
    \item (* \(F_{\text{cons}}\) and \(F_{\text{tcons}}\) contain variables, but are not under any binding. *)
    \item \(F_{\text{cons}} : \text{string} \rightarrow F_t \rightarrow F_{\text{env}} \rightarrow F_{\text{env}}\)
    \item \(F_{\text{tcons}} : \text{string} \rightarrow F_{\text{env}} \rightarrow F_{\text{env}}\)
    \item \(F_{\text{nil}} : F_{\text{env}}.\)
  \end{itemize}
\end{itemize}

\textbf{Figure 5.1}: The Coq definition of the system F language using the named binding representation.

The system F language allows for named variable binding. Simple binding is also supported for the Agda target, and the code generated is functionally identical.

Then, for every language definition with bindings, \textsc{REDEx-PLUS} also generates the substitution function, and a “closed” relation which holds when a term has no free variables (of a certain type). Both the substitution function and the “closed” relation are indexed by two non-terminal names: the first being the non-terminal whose variables are being affected, and second being the non-terminal in which these variables appear.

For example, the “substitute \(t\) in \(e\)” function in the System F language has the following signature:

\begin{itemize}
  \item \(\text{sub}_F_{\text{t\_in\_F_e}} : \text{string} \rightarrow F_t \rightarrow F_e \rightarrow F_e\)
\end{itemize}

It takes three arguments: a variable name, a \(t\) (type), and an \(e\) (expression), and
substitutes the $t$ for free variables of type $t$ inside the $e$ with that variable name. It only affects type variables in the $t$ non-terminal; term variables in the $e$ non-terminal are not affected. Notice that if it tries to replace variables of type $e$ with a $t$, it would result in a type error.

Figure 5.2 shows the definition of $\text{sub}_F_t\_in\_F_e$. Notice that it recursively calls $\text{sub}_F_t\_in\_F_t$ when encountering a subterm of type $t$ in the $F\_lam$ and $F\_tapp$ cases. Also notice that the substitution function does not recurse into the subterm in the $F\_tlam$ case if the variable to be replaced has the same name as the variable bound by $F\_tlam$. This is because substitutions should only replace free occurrences of a variable, but since all occurrences of the variable inside the subterm are bound, substitution should not happen to those variables.
5.1.1 Determining when to Generate Binding Operations

REDEX-PLUS generates a function for substituting $t$ in $e$, but it does not generate functions for, for example, substituting $e$ in $t$. This is because not all pairs of non-terminals need the substitution function. Given two non-terminals $a$ and $b$, substitution for $a$ in $b$ is only needed if $a$ may appear bound in $b$, that is, there exists some non-terminal $c$ such that

1. A binding form in $c$ binds $a$, and;
2. $b$ is $c$, or $b$ can appear as a subterm of $c$, and;
3. $a$ is $b$, or $a$ can appear as a subterm of $b$.

Figure 5.3 shows the binding relations in System F, where $a \rightarrow b$ means that “$a$ may appear bound in $b$”, indicating that a substitution function for $a$ in $b$ is needed. The rationale for the relations in System F are:

- $e \rightarrow e$ because $e$ is bound by $\text{lam}$ in $e$.
- $t \rightarrow t$ because $t$ is similarly bound by $\forall$ in $t$.
- $t \rightarrow e$ because $t$ is bound by $\text{tlam}$ in $e$, and $t$ appears in $e$ (in $\text{lam}$).

There are no outgoing arrows from $\text{env}$ as an $\text{env}$ term cannot contain a bound term as its immediate subterm. Even though $t$, which has bindings, can appear in $\text{env}$, $\text{env}$ does not have any binding forms itself, and cannot appear as a subterm in other terms with binding forms, $t$ cannot appear bound inside $\text{env}$.

5.1.2 Variable Capturing

This representation is simple, but suffers from one major drawback: if the expression being substituted with contains a variable bound in a binder, then that variable
will be captured by the binder. For example, as shown in the code example in figure 5.4, substituting the variable \( x \) with the variable \( y \) in the function \((\text{lam} \ (y : \text{nat}) \ x)\) gives \((\text{lam} \ (y : \text{nat}) \ y)\): the variable \( y \) is being captured by the lambda expression.

![Coq example](image)

**Figure 5.4:** A Coq example of variable capturing during substitution.

While not always deal-breaking, variable capturing is often undesirable, and there exists a number of possible mitigations:

- The user may require the expression being substituted with to be closed. If the expression is closed, then it contains no free variables that can get captured. The language model and the definition of substitution does not need to be changed, but extra constraints will likely be needed when expressing properties about the language. See section 7.2.2 for a lemma involving substitution that has an implicit closedness constraint.

- The substitution function may rename the variable bound at a binder to a fresh variable whenever it encounters a binder. This essentially ensures all variables bound at binders to have names not used anywhere else, so that no free variable can be captured. This approach is related to convention 5.3.4 in *Types and Programming Languages* [33, p. 71], and also similar to how substitution operates in Racket. However, coming up with fresh variable names is tricky in a proof assistant, considering a variable name not used in a term may still be present elsewhere in a theorem.

- The user may assume all variable binders to use distinct names. This assumption is often made in informal model definitions and proofs. But for

\[\text{Compute sub } "x" \text{ (F_var } "y") \text{ (F_lam } "y" \text{ F_nat (F_var } "x")\).\]

\(*\text{ computes to:}\)

\[= \text{F_lam } "y" \text{ F_nat (F_var } "y")\]

\[\text{: F_e}\]

\(*\)

\[\text{Figure 5.4: A Coq example of variable capturing during substitution.}\]
formal proofs, stating the assumption and carrying it around in proofs becomes much more difficult.

• Use a capture-avoidant binding representation, which I discuss in the rest of this chapter.

5.2 Locally Nameless

The locally nameless (LN) representation represents bound variables as de Bruijn indices. However, with LN, free variables are expressed in a more human readable way as named variables, and de Bruijn indexing is only used when dealing with bound variables. Like with de Bruijn indexing, terms written using LN are α-equivalent if and only if they are syntactically the same. The user does not need to work exclusively with numerical index, which is considered confusing and unreadable especially for larger terms [3, 8].

The tool LNgen (see 2.1.1) can be used to generate LN-based Coq definitions given a language definition in Ott.

REDEX-PLUS supports LN for all dependently typed targets. In this section I use Coq for demonstration, but similar code is generated for other dependently typed backends as well.

When generating code using LN, REDEX-PLUS first defines LN variables as a datatype in the prelude (see section 4.2), with one constructor for free variable indexed by names, and another one for bound variable indexed by a number 3. It then defines the binding operation functions for the variable datatype, and boolean equality which is needed when defining the boolean equality functions on language terms.

When translating language definitions with variable bindings, three changes are made: variable constructors take an LN variable as their sole argument; variables

\footnote{This differs from the more common approach used in the LN paper [12], where the abstract syntax of the language is modified and the language term for variables is split into one term for free and another for bound variables. I choose to introduce a distinct variable type and keep the language intact since modifying the language syntax necessitates changes to elsewhere in the language model wherever the variable constructor is used, complicating the translation process too much.}
**Figure 5.5:** The Coq definition of the LN variable type.

```coq
Inductive Ln : Type :=
| fv : string -> Ln
| bv : nat -> Ln.
```

**Figure 5.6:** The Coq definition of the system F language.

```coq
Inductive F_e : Type :=
| F_var : Ln -> F_e (* var. constructor for F_e *)
| F_app : F_e -> F_e -> F_e
| F_e_natural : nat -> F_e
| F_tapp : F_e -> F_t -> F_e
| F_tlam : F_e -> F_e (* binds F_tvar *)
| F_lam : F_t -> F_e -> F_e (* binds F_var *)

with F_t : Type :=
| F_tvar : Ln -> F_t (* var. constructor for F_t *)
| F_nat : F_t
| F___arrow__ : F_t -> F_t -> F_t
| F_forall : F_t -> F_t (* binds F_tvar *)

with F_env : Type :=
(* F_cons and F_tcons are not under any binding *)
| F_cons : string -> F_t -> F_env -> F_env
| F_tcons : string -> F_env -> F_env
| F_nil : F_env.
```

in non-terminals without bindings are turned into strings; and the variables being referred to in binding forms are removed, as the variables they bind are represented as de Bruijn indices rather than names. Figure 5.6 shows System F automatically translated into Coq by REDEX-PLUS using LN.

Next, REDEX-PLUS defines the LN binding operations: one that opens up a binding and takes the term under binding out; one that closes a term inside a binding; one that substitutes a term for a free variable; and one that binds variables.
bound at a specific level to a term \(^4\). These operations are defined as functions in the target language.

In addition, REDEX-PLUS also generates a definition of the *locally closed* property of terms. A term is considered locally closed if all bound variables within the term are bound by binders within the term. Intuitively, in a locally closed term, the level of every bound variable must be less than the number of binders in the syntax tree between the variable and the root of the syntax tree. Conversely, if a variable is bound at level greater than or equal to the number of binders, then the entire term is not locally closed. For instance, the System F expression \(F_{\text{var}}(bv\ 2)\) is not locally closed, as the bound variable is not bound by any binder outside.

Like the substitution function in the previous section, the generated LN operations as well as the locally closed property are also indexed by two non-terminal names with the same meanings: the first being the non-terminal whose variables are being affected, and second being the non-terminal in which these variables appear. Likewise, not all pairs of non-terminals need LN operations, and the process described in section 5.1.1 is used to decide which pairs do need them.

The functions carrying out the LN operations are defined as Charguéraud described [12]. An excerpt of generated code is shown in figure 5.7.

When writing a term, the user only needs to write variables using textual names, and insert “close” operations at places where the variables are bound. Similarly, REDEX-PLUS inserts LN operations in terms in judgement forms, and replace Redex substitutions with LN operations. Figure 5.8 and figure 5.9 gives examples of how LN operations are used in hand-written and REDEX-PLUS-generated code respectively.

A pitfall of LN is that because locally unclosed terms are, while undesirable, syntactically valid, the locally closed property must be stated explicitly, usually as a premise to other properties. In addition, “open”ing a binding in LN requires the new variable name given to the bound variable to be fresh so that it does not

\(^4\)Only one of \textit{sub} and \textit{bind} is needed as both of them can be implemented using the other. A proof of their equivalence is mentioned in section 7.2.3.

However, \textit{bind} has the advantage that when given an already closed term, it can perform substitution without having to first \textit{open} the term with a fresh variable and then \textit{substituting} the variable by its name. In addition to simplicity, using \textit{bind} also eliminates the need for ensuring that the variable used when \textit{opening} the term is fresh (so it would not capture any existing free variable).
with open_F_t_in_F_e x k e :=
match e with
  | F_var a0 => F_var a0
  | F_app a0 a1 => F_app (open_F_t_in_F_e x k a0) (open_F_t_in_F_e x k a1)
  | F_e_natural a0 => F_e_natural a0
  | F_tapp a0 a1 => F_tapp (open_F_t_in_F_e x k a0) (open_F_t_in_F_t x k a1)
  | F_tlam a0 => F_tlam (open_F_t_in_F_e x (S k) a0)
  | F_lam a0 a1 => F_lam (open_F_t_in_F_t x k a0) (open_F_t_in_F_e x k a1)
end

Figure 5.7: The Coq code generated for the LN operation “open $t$ in $e$”. Notice that 1) since this operation is about type variables (variables in the $F_t$ type), the $F_var$ constructor for term variables is not affected despite it taking a variable as an argument, 2) because a type variable is bound in $F_tlam$, the binding level $k$ is incremented in the recursive call, and 3) since $F_lam$ binds a term variable rather than a type variable, the binding levels are not changed in recursive calls.

clash with an existing free variable. This freshness requirement may require extra side-conditions when working with terms.

5.3 Shifted Names

Similar to the locally nameless representation, shifted names (SN) [18] uses numerical indices for bound variables and textual names for free variables. The user manipulates terms written with SN using a set of operations similar to those of LN. However, SN also affixes to each name another numerical index, in order to distinguish between distinct variables with the same name but bound at different levels. This gives the convenience of LN but removes the freshness requirements for variable names.

A Coq implementation of SN is available as the shifted-names library [19].
Definition church_true :=
F_tlam (close_F_t_in_F_e 0 "a"
(F_tlam (close_F_t_in_F_e 0 "b"
(F_lam (F_tvar (fv "a"))
(close_F_e_in_F_e 0 "t"
(F_lam (F_tvar (fv "b"))
(F_var (fv "t"))))))))).

Definition church_true_bound :=
F_tlam (F_tlam (F_lam (F_tvar (bv 1))
(F_lam (F_tvar (bv 0)) (F_var (bv 1))))).

Figure 5.8: The Church boolean “true” written using LN operations and named variables. It is equal to church_true_bound, written directly using de Bruijn indexing in bound variables.

Figure 5.9: The Redex definition and its corresponding Coq translation of the “abstract” typing rule, for $\lambda$-abstractions, in System F. Here v__a, the name of the variable bound in F_lam, is a string.
The library provides a predefined variable type, and the SN operations defined as abstract morphisms from variables to language terms. To use the library, the user needs to use the provided variable type in their language definition, define a \texttt{kleisli} function that lifts the morphisms implementing SN operations into morphisms from terms to terms, and prove (or state as Axioms) some properties of the function. The library also requires all terms to have an implicit type argument indicating how many “levels” of binding the term is in. This feature, which is not a standard part of SN, makes undesirable locally unclosed terms, which contains variables bound at a higher level than there are bindings outside, syntactically impossible.

\texttt{REDEX-PLUS} can generate Coq language definitions using the \texttt{shifted-names} library. Unfortunately, only a single variable type is supported, as having multiple variable types requires multiple level arguments in the types of terms containing bindings, which are not supported by the library. This means language models with multiple variables types, such as System F (see section 3.1.1), cannot be translated using shifted names.

When using SN, \texttt{REDEX-PLUS} first imports the needed definitions from the \texttt{shifted-names} library, and defines boolean equality for \texttt{shifted-names} variables. When translating type definitions, the extra type argument is added to all types. The extra type argument is also added to all other definitions when necessary. The \texttt{var} type from the library is used to represent variables in non-terminals with bindings and the \texttt{name} type used in ones without. Constructor arguments that are under binding will have their level arguments incremented, to indicate they are under one extra layer of binding than their parent terms. Next, \texttt{REDEX-PLUS} generates the \texttt{kleisli} function which lifts the SN operations so that the operations can be used on language terms. The \texttt{kleisli} function is required to have certain properties, which \texttt{REDEX-PLUS} defines as Axioms.

### 5.4 Higher-Order Abstract Syntax

Higher-order abstract syntax (HOAS) represents bindings in the language model using bindings in the metalanguage [21]. With HOAS, object language variables are represented as metalanguage variables, and object language binders as meta-
Inductive STLC_e { V : nat } : Type :=
| STLC_v : @var V -> @STLC_e V
| STLC_app : @STLC_e V -> @STLC_e V -> @STLC_e V
| STLC_n : nat -> @STLC_e V
| STLC_lam : @STLC_t V -> @STLC_e (S V) -> @STLC_e V
with STLC_t { V : nat } : Type :=
| STLC_nat : @STLC_t V
| STLC___arrow__ :
  @STLC_t V -> @STLC_t V -> @STLC_t V
with STLC_env { V : nat } : Type :=
| STLC_cons :
  name -> @STLC_t V -> @STLC_env V -> @STLC_env V
| STLC_nil : @STLC_env V.

Figure 5.10: The Coq definition of simply typed \( \lambda \)-calculus using shifted names. Note that the type of second argument to STLC_lam has \( (S \ V) \) instead of just \( V \) as its level argument.

language functions (in REDEX-PLUS’s case, variables and functions in the target languages respectively). There is no need to implement binding operations as the user can simply reuse existing binding features of the metalanguage.

REDEX-PLUS supports representing variable binding using HOAS for the Beluga target, in which bindings are idiomatically represented in HOAS. Currently, only language definitions are supported. The encoding in HOAS is mainly carried out in the following two steps: Before generating the Poultry definitions, variable constructors are removed from non-terminal definitions. Variable constructors are no longer needed in the syntax definition as they represent variables in the object language, which, under HOAS, are represented by metalanguage variables instead. And then, when generating the signatures of constructors with binders, arguments under binding are required to have a function type from the type being bound to the type of the term itself. Figure 5.11 shows the generated Beluga definition for the simply typed \( \lambda \)-calculus.

55
STLC_e : type.
STLC_t : type.

STLC_app : STLC_e → STLC_e → STLC_e.
STLC_n : nat → STLC_e.
STLC_lam : STLC_t → (STLC_e → STLC_e) → STLC_e.
STLC_nat : STLC_t.
STLC___arrow__ : STLC_t → STLC_t → STLC_t.

**Figure 5.11:** The Beluga definition of simply typed λ-calculus using HOAS. Notice that the constructor for term variable has been removed.
Chapter 6

Implementation

At its core, REDEX-PLUS is a compiler from Redex statements (in the supported subset) to target proof assistant language. In this chapter, I describe REDEX-PLUS’s interface and its overall structure, as well as the main translation procedure.

6.1 Interface

REDEX-PLUS is implemented as a Racket library. To use REDEX-PLUS, a user requires REDEX-PLUS in their Racket code the same way they require other libraries. require’ing REDEX-PLUS makes available all of Redex except its GUI tool 1, that is, all of redex/reduction-semantics and redex/pict. REDEX-PLUS also provides a handful of macros for configuring REDEX-PLUS, allowing the user to choose the translation target, binding representation, variable representation, and metafunction and reduction relation representations as they wish. Figure 6.1 shows how REDEX-PLUS is required and configured to generate Coq code using the locally nameless representation.

The user then writes down the language model using the provided Redex macros. Like any other Redex model, REDEX-PLUS language models are executable Racket programs, and running a REDEX-PLUS model prints the generated target code to the standard output. The user is responsible for making sure that only the supported

---

1The Redex GUI tool (from redex/gui) is not re-provided because GUI features are not supported by certain development environments. Users with GUI-compatible environments may import redex/gui separately in their code.
subset of Redex is used in defining the model. Otherwise, the output is undefined.

For language models using only Redex features within the supported subset, REDEX-PLUS can be used as a drop-in replacement of Redex: the user can access REDEX-PLUS features by simply require`ing REDEX-PLUS instead of Redex, and configuring REDEX-PLUS as needed. On the other hand, all REDEX-PLUS language models are also valid in Redex.

```racket
#lang racket

(require redex-plus)

(use-backend coq)
(use-binding locally-nameless)

(define-language F
  ; non-terminal definitions
)

; other definitions follow

Figure 6.1: require`ing the REDEX-PLUS library, configuring it, and using it to define system F.

6.2 Internal Workings

REDEX-PLUS’s core compilation procedure consists of the following three steps:

1. A statement in a language model, which is a valid Racket syntax object, is parsed using the syntax/pars e library. Statements like define-language and define-judgment-form, which provides multiple ways to define the same non-terminal or judgement rule, are also normalized in this stage. After this stage, each statement in the language model is turned into a Racket
data structure which is then passed to the next stage. The parsing stage is completely independent of the translation target.

2. The statement is passed down a number of compilation passes. Together, the passes carry out the transformation described in section 4.1. A more detailed explanation of the passes is given in section 6.3.

The exact passes a model goes through depend on the target proof assistant, although the majority of the passes are shared between all targets. The final pass produces definitions in a target-agnostic intermediate representation named Poultry, described in further detail in section 4.1.1.

3. The intermediate Poultry statements are translated into concrete target code and written to the standard output. The detailed translation process is described in section 4.2. The translation is done in one single step, and little could be shared between the translation process for different targets. This step does not only translate the definitions themselves, but also generate the boilerplate code, such as boolean equalities and binding operations, required to use the language model.

During macro expansion, each statement such as define-language is wrapped in a call to the compilation procedure outlined above. The compilation procedure is run, and target code generated, during the run-time of the Racket program.

6.3 The Compilation Passes

The transformation from Redex statements to Poultry definitions is carried out by a sequence of passes. Each pass either checks, records, or modifies the statement data structure, and passes the modified data structure down to the next pass. The final pass produces target-agnostic, intermediate Poultry definitions which are then translated into target code.

The decision to use multiple smaller passes instead few large, monolithic passes is inspired by “nanopass” compilers where each of the numerous “nanopasses” does only a single task [25, 36], albeit some of REDEX-PLUS’s passes do more than one thing. This approach allows great flexibility in the translation process:
some passes can be bypassed or added depending on user configurations; the final pass can be replaced to generate a different type of Poultry definition; and any new translation step can be added as new passes, with little impact to existing passes, as long as it accepts and outputs compatible models.

Table 6.1 shows the translation pipelines and their passes for all supported statements. While the translation procedure for different statements differ significantly, all pipelines contains the following types of passes: passes that check for errors and missing informations; a pass that normalizes non-terminal aliases to canonical non-terminal names as defined in section 4.1.2; a pass whose name starts with simplify-, in which terms and patterns in the statements written in the concrete syntax of the object language are simplified into the abstract syntax (as described in section 4.1.2); and the final pass which generates Poultry statements.

define-language Passes

- The first two passes checks for common mistakes in the language definitions and aborts the translation if any is found. check-language-well-foundedness ensures that recursively-defined non-terminals have non-recursive “base cases”, without which one cannot write concrete terms, and the generated language definition becomes invalid in some targets; detect-misspelled-non-terminals checks if a symbol in a non-terminal definition is a likely misspelling of an existing non-terminal name, or is missing an underscore\(^2\). Such errors will otherwise be carried silently into future translation steps and cause errors that are difficult to debug.

- display-prelude prints the prelude (see section 4.2) to the output.

- The passes whose names begin with save- or register- stores information about the language for future uses. The saved information is retained after the language definition has been translated as other definitions still use them. Apart from register-and-remove-variable-non-terminals, which removes non-terminals used to solely represent typed variables, most of these passes do not make any changes to the model.

- simplify-constructors simplifies constructors as described in section 4.1.2. This step also generates “simplifiers” which are used to simplify terms and patterns in other statements of the language model in the future.

---

\(^2\) In a language with a non-terminal named e, for example, e₁ in non-terminal definition matches a term of the non-terminal e, but e₁ would be a literal.
After this pass, all terms are in the normalized and simplified form.

- **implement-binding-form** writes binding information into terms via the following transformations: variable constructors in non-terminals with binding are marked as “bound”; variables referred to in binding forms are removed; and terms under binding in binding forms are marked as “affected”. Non-terminals without bindings are not changed.
  After this pass, all bound variables would be represented in the aforementioned form.

- **order-constructors** places non-terminals containing binders and bound variables first.

- **encode-in-hoas** takes language definitions with binding information already written in and turns them into concrete HOAS representations.

### define-judgement-form Passes

- **check-contract-existence** checks if a contract, which states the non-terminal types of arguments to this judgement form, exists. If there is no contract, an error is raised as REDEX-PLUS cannot deduce a contract on its own.

- **generate-default-rule-names** generates names for unnamed rules.

### define-metafunction-form Passes

- **replace-booleans** replaces all Racket boolean literals in metafunction definitions with fresh symbols to prevent any potential clashes with non-terminal patterns of the same names.

- **translate-metafunction** translates the function into its Poultry representation.

- **optimize-matches** simplifies unnecessary #:match expressions in the Poultry representation of a metafunction.

### reduction-relation Passes

- The first two passes, as names suggest, *deduces* the domain and codomain of the relation if they are not specified.
• Depending on user setting, the reduction relation will be represented as either a relation (similar to judgement forms) or a function. (similar to metafunctions; executable in the target language).

The two passes named `reduction-relation->...-definition` translates reduction relations into the respective Poultry statements. This pass also saves the Poultry definition generated, so it can be later used to generated the closure of the reduction.

• For proof assistant targets with support for defining custom notations or operators, `define-notation` generates the notation definition.

### 6.4 Adding New Translation Targets

To add a new translation target, or to implement a new binding representation, one needs to carry out the following steps:

1. Implement the translation from normalized, simplified object language terms in Redex into the target language. If the target makes use of types, then the translation from these two procedures should be implemented first as they will be used in the translation of almost all other definitions.

   In existing translation implementations, these processes are carried out by the `translate-term` and `translate-type` procedures respectively.

2. Implement the translation from Poultry definitions into the target language. At the minimum, translation of `define-type` statements must be implemented in order to represent the syntax of the target language; but other definitions are not necessary.

3. Decide if the translation pipeline needs to be modified. If the target language is based on a logic system similar to that of an existing target, then there is a good chance that it can reuse most, if not all, passes already implemented for the existing target. For example, if one wishes to add a dependently-typed target using the locally nameless representation, then it is unlikely that they will need a pipeline significantly different from the existing pipelines for Coq and Agda.
Table 6.1: Compilation passes for all REDEX-PLUS statements and targets. “○” means this pass is a part of the translation process for the target; “|” means it is not; “△” means it depends on user setting; “⊕” means only one of the passes marked as so is included, depending on user settings; and “×” means this statement is not supported by the target. The targets (Coq, Agda, Beluga, and SMT-LIB) are abbreviated to their initial letters.

<table>
<thead>
<tr>
<th>Pass</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>check-definition-well-foundedness</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>detect-misspelled-non-terminals</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>display-prelude</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>save-language-definition</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>register-if-has-binding</td>
<td>○</td>
<td>○</td>
<td></td>
<td>○</td>
</tr>
<tr>
<td>register-patterns</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>register-and-remove-variable-non-terminals</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>simplify-constructors</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>register-constructors</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>implement-binding-form</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td></td>
</tr>
<tr>
<td>register-variable-constructors</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td></td>
</tr>
<tr>
<td>register-substitution</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td></td>
</tr>
<tr>
<td>order-constructors</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td></td>
</tr>
<tr>
<td>encode-in-hoas</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>translate-language-definition</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td></td>
</tr>
</tbody>
</table>

(a) Compilation passes for define-language.

<table>
<thead>
<tr>
<th>Pass</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>check-contract-existence</td>
<td>○</td>
<td>○</td>
<td>×</td>
<td>○</td>
</tr>
<tr>
<td>normalize-contract-types</td>
<td>○</td>
<td>○</td>
<td></td>
<td>○</td>
</tr>
<tr>
<td>generate-default-rule-names</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td></td>
</tr>
<tr>
<td>simplify-rule-patterns</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td></td>
</tr>
<tr>
<td>translate-judgement-form-definition</td>
<td>○</td>
<td>○</td>
<td></td>
<td>○</td>
</tr>
</tbody>
</table>

(b) Compilation passes for define-judgement-form.

<table>
<thead>
<tr>
<th>Pass</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>normalize-signature-types</td>
<td>○</td>
<td>○</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>replace-bools</td>
<td>○</td>
<td>○</td>
<td></td>
<td></td>
</tr>
<tr>
<td>register-mf</td>
<td>○</td>
<td>○</td>
<td></td>
<td></td>
</tr>
<tr>
<td>simplify-patterns</td>
<td>○</td>
<td>○</td>
<td></td>
<td></td>
</tr>
<tr>
<td>replace-wildcards</td>
<td>○</td>
<td>○</td>
<td></td>
<td></td>
</tr>
<tr>
<td>translate-metafunction</td>
<td>○</td>
<td>○</td>
<td></td>
<td></td>
</tr>
<tr>
<td>optimize-matches</td>
<td>○</td>
<td>○</td>
<td></td>
<td></td>
</tr>
<tr>
<td>generate-metafunction-definition</td>
<td>○</td>
<td>○</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) Compilation passes for define-metafunction.
4. Add new passes, or make modified versions of existing passes, if needed. Since there are almost certainly some existing passes in the new pipeline, one must make sure that the output of the new passes are compatible with the subsequent existing passes.

5. Unless one is expanding REDEX-PLUS’s translatable subset, there is little reason to change the syntax parser.
Chapter 7

Evaluation

As stated in section 1.3, the goal of REDEX-PLUS is to allow users to write language models in Redex, and use REDEX-PLUS to translate the models to proof assistant languages in which proofs can then be written, without having to manually define the language models in a proof assistant language, or to write any boilerplate. In this chapter, I show that REDEX-PLUS fulfills the aforementioned goal. First, I show that REDEX-PLUS can translate the same language model using different binding representations to different targets, and save the user from having to write large amounts of boilerplate; then, I show that REDEX-PLUS generates language definitions that can be used to prove non-trivial properties of language models. Finally, I discuss some findings about the SMT-LIB translation target and its limitations.

7.1 Automatic Code Generation

In this section, I evaluate REDEX-PLUS’s ability to generate code targeting different proof assistants using different binding representations. Here, I look at two language models:

- The Simply Typed Lambda Calculus (STLC), whose Redex model has 8 lines of language definition and 50 lines of other definitions, excluding imports and configurations. The STLC model is used for evaluation proofs later in this chapter in section 7.2.
Table 7.1: Breakdown of the amount of generated code, in terms of lines of code.

- The System F, whose Redex model has 18 lines of language definition and 63 lines of other definitions. System F, introduced in section 3.1.1, is used as an example throughout this thesis.

I translate the two models into Coq and Agda, using both the named and locally nameless binding representations. Then, for each of the generated code file, I count the number of lines of code in each of the following categories:

**Prelude (P)** Library imports, and definitions of variable types and operations on them, all located at the beginning of the file.

**Language Definition (LD)** Data definition of the language’s abstract syntax.

**Language Boilerplate (LB)** Boilerplate for the language definition. Mostly the definition of boolean equality.

**Binding Operations (BO)** The substitution function, and operations such as “open” and “close” for the locally nameless representation.

**Other Definitions (OD)** Definitions of judgement forms, metafunctions, and reduction relations.

**Other Boilerplate (OB)** Boilerplate related to the other definitions, such as autoderive tactic definitions in Coq.

Table 7.1 and figure 7.1 shows the numbers of lines of code that falls under each category for each target, language model, and binding representation.
7.1.1 Discussion

First of all, it is worth noting that the two Redex language models used in this section are target-agnostic: they do not include any target-specific information. In addition, no changes to the language models are needed when changing the target proof assistant or the binding representation. These two observations show that Redex language models are portable: the user needs to only write the model once, and Redex-Plus is able to translate the model to different targets using different binding representations without having to modify the language models.

As shown by the breakdown of code content, significant proportions of the code generated are binding operations, sometimes over 40%. Binding operations also take more lines of code to define when there are more than one type of variables, which is the case with System F, or when using the locally nameless representa-
tion, which has more binding operations. In contrast, language definition proper uses much fewer lines of code. By automatically generating binding operations as well as other boilerplates, REDEX-PLUS lets the user focus on the language definitions themselves, without having to go through the tedious, error-prone process of writing all the binding operations by hand.

7.2 Properties of the Simply Typed Lambda Calculus

```lisp
(define-language STLC
  [t ::= nat (-> t t)]
  [e ::= (v variable) (lam (variable : t) e) (app e e) natural]
  [env ::= (cons (variable : t) env) nil]
  #:binding-forms
  (lam (variable : t) e #:refers-to variable))

(define beta-single-step
 (reduction-relation
  STLC
  #:domain e
  [--> (app (lam (variable : t) e_b) e_a)
     (substitute e_b variable e_a) "subst"]
  )
)

(define beta (compatible-closure beta-single-step STLC e))

Figure 7.2: The Redex language definition of STLC and $\beta$-reduction.

The Simply Typed Lambda Calculus (STLC) is a simple language model often used as a basis for developing more sophisticated languages. In this section, I prove some important and well-known properties about typing and reduction in an extension of STLC that includes natural numbers. The language model definition as well as the proofs are largely based on chapters Lambda and Properties of Programming Language Foundations in Agda [40], and some of the proofs’ struc-
tures resemble those found in Volume 2 of Software Foundations [34]. The proofs are done in Coq and Agda using language models translated by REDEX-PLUS from the Redex definition using the “named” binding representation described in section 5.1. Figure 7.2 shows the Redex definition of the STLC syntax and $\beta$-reduction, and figure 7.3 shows the typing judgement as well as related definitions.

There are a few notable differences between the STLC language model used here and more “basic” definitions of STLC such as the one in Types and Programming Languages [33, p. 103]:

- Natural numbers (natural) are valid expressions. There is also a natural number type (nat), and an associated typing rule (“natural”). The natural number type serves as a necessary “base” type for the type system. If the language does not have a base type, it would be syntactically impossible to write a well-formed type, and no expression in the language can be well-typed at all.

- The Value judgement form determines whether or not an expression is considered a “value”. In many definitions, “value” is defined as a syntactic construct that is a subset of expressions, rather than a relation or a property.

- The typing environment is expressed as an explicit, nil-terminated list constructed by cons’ing type statements of the form “variable $v$ has type $t$” to another environment. The “$\text{In}$” judgement is used to “look up” the type statement associated with a variable in the environment, “returning” only the first matching statement and discarding the rest. If a typing statement stating a variable has a certain type In an environment, then, by the “context” typing rule, the variable, as an expression, has that type given the environment.

- $\beta$-reduction (beta) is defined as the compatible closure of the reduction of function application (beta-single-step). This means reduction can happen in any order, and the body sub-expression in an lambda abstraction can reduce on its own. Fortunately, for all but the last proof, allowing lambda abstractions to reduce on their own does not matter, and the compatible closure is all I need to prove the theorems. For the final proof, reducing the
(define-metafunction STLC
  diff : variable variable -> boolean
  [(diff variable_1 variable_1) #f]
  [(diff variable_1 variable_2) #t])

(define-judgement-form STLC
  #:contract (Value e)
  [(Value natural) "natural-is-value"]
  [(Value (lam (variable : t) e)) "lam-is-value"])

(define-judgement-form STLC
  #:contract (In env variable t)
  [(In (cons (variable : t) env_t) variable t) "axiom"]
  [(In env variable_1 t_1)
   (side-condition (diff variable_1 variable_2))
   --------------------------------------------- "cons"
   (In (cons (variable_2 : t_2) env) variable_1 t_1)])

(define-judgement-form STLC
  #:contract (T env e t)
  [(In env variable t)
    ------------------- "context"
    (T env (v variable) t)]
  [--------- "natural"
    (T env natural nat)]
  [(T (cons (variable_a : t_a) env_t) e_b t_b)
    ------------------------------------------- "abstract"
    (T env_t (lam (variable_a : t_a) e_b) (-> t_a t_b))]
  [(T env e_f (-> t_1 t_2)) (T env e_a t_1)
    ------------------------------- "apply"
    (T env (app e_f e_a) t_2)])

Figure 7.3: The Redex definition of the typing judgement in STLC.
Notation "$ e >-> e' $" := (STLC_beta e e') (at level 20).
Notation "$ env |- e : t $" := (STLC_T env e t)
(at level 20, e at next level, t at next level).
Notation "$ x : t :: env $" := (STLC_cons x t env)
(at level 20, t at next level, env at next level).
Notation "$ x : t <- env $" := (STLC_In env x t)
(at level 20, t at next level, env at next level).

Table 7.4: Notation definitions for STLC, in Coq and Agda.

\[
\begin{align*}
[\_\Rightarrow\_] &= \text{sub-STLC-e-STLC-e} \{ [ v \Rightarrow e1 ] e2 = \text{sub } v e1 e2 \} \\
\_\Rightarrow\_ &= \text{STLC-beta} \\
\_\in\_\in\_ &= \text{STLC-cons} \{ - v \in t :: env = (\text{cons } (v : t) \text{ env}) \} \\
\emptyset &= \text{STLC-nil} \\
\_\not\in\_\in\_ &= \text{STLC-In} \{ - v \not\in t \in \text{env} = (\text{In } v t \text{ env}) \} \\
\_\vdash\_\in\_ &= \text{STLC-T} \{ - \text{env } \vdash e \in t = (\text{T env e t}) \} \\
\end{align*}
\]

Body sub-expressions of lambda abstractions does cause problems. I discuss
the problem and my workaround in detail later in section 7.2.3.

In Redex, I can use a context closure to forbid the reduction of certain sub-
terms, and to impose a reduction order. However, context closures are not
supported by REDEX-PLUS. Alternatively, I can also define the entire reduc-
tion relation manually as a judgement form, but that would not be idiomatic
in Redex.

For the sake of clarity and brevity, I manually defined a number of notational
shorthands as shown in figure 7.4, and used them in the proofs.

The Redex language models used can be found in the code repository in examples/
coq-stlc-named-explicit.rkt and examples/agda-stlc-named-explicit.rkt.
The completed proofs can also be found in code repository inside the proofs di-
rectory.
7.2.1 Progress

**Theorem 1** (Progress). For any expression $e$ and type $t$, if $e$ has type $t$ under an empty context (i.e. $(\text{T nil } e \ t)$), then either $e$ is a value (i.e. $(\text{Value } e)$), or there exists an expression $e_2$ such that $e$ reduces to $e_2$.

In Coq:

```coq
Theorem Prog : forall e t, 
STLC_nil |- e : t -> is_value e / exists e', e --> e'.
```

In Agda:

```agda
prog : \forall e t -> 
\emptyset |- e : t \rightarrow \text{STLC-Value } e \lor \exists e' (e \Rightarrow e')
```

*Proof.* By induction on the typing relation $(\text{T nil } e \ t)$. Most induction cases are either vacuous because the typing property does not hold, or trivial as the expression is already a value. In the application case, the proof proceeds by case analysis on the induction hypothesis of the first sub-expression. If the sub-expression is a value, then it must be an lambda abstraction for the expression to be well-typed, in which case the whole expression undergoes $\beta$-reduction; otherwise, it reduces to some other expression, and by the compatible closure the entire expression also reduces.

See section A.1.1 for the Coq proof, and section A.2.1 for the Agda proof. □

7.2.2 Preservation of Types under Substitution

**Lemma 1** (Preservation of Types under Substitution (Substitution Lemma)). For any environment $env$, variable $x$, type $xt$, expression $v$ and $e$, and type $t$, if $v$ has type $xt$ under an empty context (i.e. $(\text{T nil } v \ xt)$), and $e$ has type $t$ under $(\text{cons } (x : xt) \ env)$ (i.e. $(\text{T } \text{cons } (x : xt) \ env \ e \ t)$), then $e$ with all free occurrences of $x$ substituted with $v$ has type $t$ under $env$ (i.e. $(\text{T } env \ (\text{substitute } x \ v \ e \ t)$).
In Coq:

\[ \text{Lemma Subst :} \]
\[ \forall \text{env x xt v e t}, \]
\[\text{STLC\_nil |- v : xt} \]
\[\rightarrow x : xt :: \text{env} |- e : t \]
\[\rightarrow \text{env |- sub x v e : t}. \]

In Agda:

\[ \text{subst :} \forall \{ \text{env v e to vt t} \} \]
\[\rightarrow \emptyset |- \text{to} \in \text{vt} \]
\[\rightarrow (v \in \text{vt :: env}) |- e \in t \]
\[\rightarrow \text{env |- [ v } \Rightarrow \text{to } ] e \in t \]

Notice that the first premise in the hypothesis, which states that the expression \(v\) has type \(xt\) under a \text{nil} context, implies the closedness of \(v\). If \(v\) contains any free variable, then it would not have a type.

**Proof.** By induction on either the expression \(e\) and then inversion of the typing judgement \(T (\text{cons} (x : xt) \text{ env}) e t\). Alternatively, by induction on the typing judgement directly.

See section A.1.3 for the Coq proof, which uses the former approach in order to obtain a sufficiently strong induction hypothesis, and section A.2.3 for the Agda proof, which uses the later approach.

The proof of lemma 1 makes use of a number of auxiliary lemmas. Most of them are about the preservation of types under changes to the typing environment. The rest are about properties about the \text{diff} function. The auxiliary lemmas are states and proven in sections A.1.2 and A.2.2 in Coq and Agda respectively.

### 7.2.3 Preservation

For this theorem instead of using the automatically generated \text{beta} reduction relation which has some peculiarities as discussed earlier, I use the hand-written definition of \(\beta\)-reduction shown in figure 7.5 instead. The only difference between the hand-written definition and the automatically generated one is that the hand-written definition does not allow the body sub-expression in lambda abstractions to reduce. That reduction rule is problematic because it leads to a case where the induction hypothesis is too weak, and strengthening the induction hypothesis would require a stronger substitution lemma that is significantly more difficult to prove.
It is worth noting that this particular difficulty comes from theorem proving, rather than REDEX-PLUS itself, and is thus orthogonal to the evaluation of REDEX-PLUS.

\[
\text{Inductive } \text{beta} : \text{STLC}_e \rightarrow \text{STLC}_e \rightarrow \text{Prop} :=
\begin{align*}
| \text{beta\_single\_step\_subst\_cc} \ (\text{variable} \ t \ e\_b \ e\_a) : & (\text{beta} \ (\text{STLC\_app} \ (\text{STLC\_lam} \ \text{variable} \ t \ e\_b) \ e\_a) \\
& (\text{sub} \ \text{variable} \ e\_a \ e\_b)) \\
| \text{beta\_cc\_app} \ (e\_from \ e\_to \ e\_i1) : & (\text{beta} \ e\_from \ e\_to) \\
& (\text{beta} \ (\text{STLC\_app} \ e\_from \ e\_i1) \ (\text{STLC\_app} \ e\_to \ e\_i1)) \\
| \text{beta\_cc\_app\_2} \ (e\_from \ e\_to \ e\_i0) : & (\text{beta} \ e\_from \ e\_to) \\
& (\text{beta} \ (\text{STLC\_app} \ e\_i0 \ e\_from) \ (\text{STLC\_app} \ e\_i0 \ e\_to)).
\end{align*}
\]

\text{Notation} "t1 \rightarrow t2" := (\text{beta\_reduction} t1 \ t2) (at level 20).

\[
\text{data} \ _\rightarrow_\_ : \text{STLC}\_e \rightarrow \text{STLC}\_e \rightarrow \text{Set}
\begin{align*}
\text{data} \ _\rightarrow_\_ \ &\text{where} \\
\beta \rightarrow : & \forall \ (\text{``variable} \ t \ e\_b \ e\_a) \\
& \rightarrow \text{STLC\_Value} \ e\_a \\
& \rightarrow ((\text{STLC\_app} \ (\text{STLC\_lam} \ \text{variable} \ t \ e\_b) \ e\_a) \\
& \rightarrow (\text{sub}\_\text{STLC\_e}\_\text{STLC\_e} \ e\_b \ e\_a)) \\
\text{app}\_\rightarrow^\_1 : & \forall \ (e\_from \ e\_to \ e\_i1) \\
& \rightarrow (e\_from \rightarrow e\_to) \\
& \rightarrow ((\text{STLC\_app} \ e\_from \ e\_i1) \rightarrow (\text{STLC\_app} \ e\_to \ e\_i1)) \\
\text{app}\_\rightarrow^\_r : & \forall \ (e\_from \ e\_to \ e\_i0) \\
& \rightarrow \text{STLC\_Value} \ e\_i0 \rightarrow (e\_from \rightarrow e\_to) \\
& \rightarrow ((\text{STLC\_app} \ e\_i0 \ e\_from) \rightarrow (\text{STLC\_app} \ e\_i0 \ e\_to))
\end{align*}
\]

Figure 7.5: Definitions of $\beta$-reduction and the associated notation in Coq and Agda. The rule for reducing the body of an lambda abstraction has been removed, and the left sub-expression of a function application is not required to be a value before the right sub-expression could reduce.

\textbf{Theorem 2} (Preservation). \textit{For any expression $e\_1$ and $e\_2$, and type $t$, if $e\_1$ has type $t$ under the empty context (i.e. $(T \ \text{nil} \ e\_1 \ t)$, and $e\_1$ reduces to $e\_2$, then $e\_2$ has type $t$ under the empty context (i.e. $(T \ \text{nil} \ e\_2 \ t)$).}
In Coq:

```coq
Theorem Pres : forall e t e',
  STLC_nil |- e : t ->
  e --> e' ->
  STLC_nil |- e' : t.
```

In Agda:

```agda
pres : ∀ {e t e'}
    → ∅ ⊢ e ∈ t
    → e --> e'
    → ∅ ⊢ e' ∈ t
```

**Proof.** By induction on the typing relation STLC_nil |- e : t. Most induction cases are vacuous or trivial. The only interesting case is that of the typing of function application, which requires the substitution lemma to prove.

See section A.1.4 for the Coq proof, and section A.2.4 for the Agda proof.

7.2.4 Discussion

By proving some important properties of STLC, I show that REDEX-PLUS can generate language models suitable for use in practical theorem-proving in multiple proof assistant languages. The proofs follow the conventional styles in the target proof assistants, and are generally easy to work with. For the Coq proof, the autoderive tactic, which automatically searches for the judgement form derivation that proves a goal, has proven useful as it was used multiple times throughout the proofs.

While most of the theorems (including the auxiliary lemmas) in this section do not require any additional definitions to be manually added to the language model to prove, I still needed to manually define β-reduction as shown in figure 7.5. This can be avoided if Redex’s context-closures are supported in the future. In addition, while not strictly required, I still added the notation definitions in figure

75
7.4 for better clarity. A future version of REDEX-PLUS may be able to generate the notation definitions on its own, or, in the case of Agda, follow the original language syntax closer when generating language definitions.

### 7.3 Unsuccessful Proof Attempts

I made two other attempts to prove properties of language models using REDEX-PLUS-generated code. However, due to the difficulty of the proofs and time constraints, I did not finish either of them. I discuss these unsuccessful attempts in this section to analyze whether these attempts reveal limitations in REDEX-PLUS or not.

#### 7.3.1 Properties of STLC using the Locally Nameless Binding

I tried to prove the same properties of STLC discussed in section 7.2 using the locally nameless binding representation instead of the named representation. I was stuck when proving the substitution lemma. I believe additional lemmas about locally nameless binding operations would have helped me proceed with the proof. One possible future extension to REDEX-PLUS is to generate these lemmas automatically, which I discuss in section 8.2.

In comparison, the LNgen (see section 2.1.1) authors have successfully proven similar properties of STLC using a locally nameless version of STLC generated by LNgen [2]. Notice that LNgen generates many binding lemmas, suggesting that these lemmas are useful when writing proofs and REDEX-PLUS would benefit from being able to generate them.

#### 7.3.2 Properties of System F

I also attempted to prove the same properties of System F, using the named binding representation. For System F, the final type preservation proof requires not only the (term) substitution lemma, but also a similar lemma about substituting a type variable with a type:

**Lemma 2** (Type Substitution Lemma). For any environment $\text{env}$, type variable $\text{tv}$, expression $e$, and types $t_1$ and $t_2$, if all type variables bound within $e$, $\text{env}$, and
t_2 are distinct, and e has type t_1 under (tcons tv env) (i.e. (T (tcons tv env) e t_1)), then e with tv substituted by t_2 has type t_1 with tv substituted by t_2 under the context env with tv substituted by t_2 (i.e. (T (substitute tv t_2 env) (substitute tv t_2 e) (substitute tv t_2 t_1))).

Notice the premise requiring bound type variables to be distinct \(^1\). This premise turned out to be difficult to formally express, and I was stuck when proving the type substitution lemma. It is unlikely that this proof can be made easier by improvements to REDEX-PLUS, as the difficulty in this proof lies in mechanizing the notion of “distinctness” in a proof assistant.

### 7.4 Automatic Solving with SMT Solvers

REDEX-PLUS supports generating language definitions in the SMT-LIB 2 language for automatic theorem proving using an SMT solver (see section 4.2.3). Since REDEX-PLUS does not support translating variable bindings into SMT-LIB, I will use an alternative language model of simple arithmetic expressions, shown in figure 7.6 for demonstration.

I can then use the SMT solver to automatically prove that certain expressions reduce to other expressions. Here, in order to prove that a relation holds, I first assert that the relation does not hold, and then ask the SMT solver to check the satisfiability of the statement. If the statement is unsatisfiable, then it means the statement must be true. Running the code in figure 7.7 returns (unsat) two times, which means in both cases, the reduction relation holds.

#### 7.4.1 Limitations of SMT Solvers

Unfortunately, there are some serious limitations around using SMT language models generated by REDEX-PLUS, namely, the SMT solver making up non-existent derivation rules when finding solutions, and difficulty with inductive reasoning.

Consider the code in figure 7.8, which attempts to find x, which is what the \texttt{Arith} (see figure 7.6) expression “(add 1 1)” reduces to. Surprisingly, Z3 thinks x can be zero, despite the derivation rules for the reduction relation say otherwise.

\(^1\)See section 5.1.2 for discussion on a related requirement.
(define-language Arith
  [e ::= (add e e) (min e e) (mul e e) integer])

(define eval-single-step
  (reduction-relation
   Arith
   #:domain e
   #:codomain e
   [--> (add integer_1 integer_2)
    (, (+ (term integer_1) (term integer_2))) "add"]
   [--> (min integer_1 integer_2)
    (, (- (term integer_1) (term integer_2))) "min"]
   [--> (mul integer_a integer_b)
    (, (* (term integer_a) (term integer_b))) "mul"]))

(define eval (compatible-closure eval-single-step Arith e))

**Figure 7.6:** The Redex definition of the a simple arithmetic expression language and its reduction relation.

This happens because Arith-eval, the reduction relation, is represented as an uninterpreted function in SMT-LIB. The SMT solver is free to come up with a definition for the function however it wants, as long as it satisfies the assertions about the function. This problem makes it impossible to use SMT solvers to find terms that satisfy a certain property, limiting their use to checking the correctness of statements only.

Also consider the language model of natural numbers, the “greater or equal to” relation, and the “twice as large as” relation, as defined in figure 7.9. Figure 7.10 shows the SMT code for checking the correctness of two simple theorems regarding this language model: that all natural numbers are greater or equal to zero; and that if two numbers are both twice as large as another number, then the two numbers are the same. Proving either theorem requires induction over the natural number datatype. When given this code as input, Z3 runs indefinitely without giving any results, and CVC5, with the --quant-ind flag turned on, can prove the
\[ (\text{mul} \ (\text{add} \ 10 \ 20) \ (\text{min} \ 20 \ 10)) \rightarrow (\text{mul} \ 30 \ (\text{min} \ 20 \ 10)) \]

\text{(push)}

\text{(assert \ (not \ (Arith-eval \ (Arith-mul \ (Arith-add \ (Arith-e-integer \ 10) \ (Arith-e-integer \ 20)) \ (Arith-e-integer \ 30) \ (Arith-min \ (Arith-e-integer \ 20) \ (Arith-e-integer \ 10)))))})

\text{(check-sat)}

\text{(pop)}

\[ ; \ (\text{mul} \ (\text{add} \ 10 \ 20) \ (\text{min} \ 20 \ 10)) \rightarrow (\text{mul} \ (\text{add} \ 10 \ 20) \ 10) \]

\text{(push)}

\text{(assert \ (not \ (Arith-eval \ (Arith-mul \ (Arith-add \ (Arith-e-integer \ 10) \ (Arith-e-integer \ 20)) \ (Arith-e-integer \ 10)))))}

\text{(check-sat)}

\text{(pop)}

**Figure 7.7:** SMT-LIB code for proving two reduction relations. The “push” and “pop” commands are for delimiting each of the two proofs.

First theorem but not the second. SMT solvers’ difficulty with inductions is well-known [28, 35]. Given language models often involve inductively defined syntax structures, this problem with inductive proofs will likely become an obstacle when using SMT solvers with such language models.
(declare-fun x () Arith-e)
(assert
 (Arith-eval (Arith-add (Arith-e-integer 1) (Arith-e-integer 1)) x))
(check-sat)
(get-model)

sat
(model
 (define-fun x () Arith-e
  (Arith-e-integer 0))
 (define-fun Arith-eval-single-step
  ((x!0 Arith-e) (x!1 Arith-e)) Bool
   true)
 (define-fun Arith-eval ((x!0 Arith-e) (x!1 Arith-e)) Bool
   (let ((a!1 (= (ite (= x!1 (Arith-e-integer 0))
                   (Arith-e-integer 0)
                   (Arith-e-integer 2))
                   (Arith-e-integer 0)))
     (a!2 (= (ite (= x!1 (Arith-e-integer 0))
                   (Arith-e-integer 0)
                   (Arith-e-integer 2))
                   (Arith-e-integer 2))))
   (or a!1 a!2)))
)

Figure 7.8: SMT-LIB code for finding what the expression \(\texttt{add 1 1}\) reduces to, and the Z3 output from running the code. Notice that the model Z3 came up with states that \texttt{Arith-eval-single-step} is always true (i.e. every term reduces to every other term), and \texttt{Arith-eval} is true as long as the reduction result is zero or two, which are both clearly nonsensical.
(define-language Nat
  [n ::= (S n) Z])

(define-judgement-form Nat
  #:contract (geq n n)
  [(geq Z Z) "zero"]
  [(geq (S n_1) n_2) (geq n_1 n_2) "succ"]
  [(geq (S n_1) (S n_2)) (geq n_1 n_2) "succ2"])

(define-judgement-form Nat
  #:contract (twice n n)
  [(twice Z Z) "identity"]
  [(twice (S (S n_1)) n_2) (twice n_1 n_2) "twice"])

Figure 7.9: The Redex definition of natural numbers and two relations.

(push)
(assert (not (forall ((n Nat-n)) (Nat-geq n Nat-Z))))
(check-sat)
(pop)

(push)
(assert (not (forall ((n Nat-n) (n21 Nat-n) (n22 Nat-n))
  (= (Nat-twine n21 n) (Nat-twice n22 n) (= n21 n22))))))
(check-sat)
(pop)

Figure 7.10: SMT-LIB code for checking the correctness of two simple theorems about natural numbers.
Chapter 8

Conclusion

8.1 Contributions

In this thesis I presented REDEX-PLUS, an implementation of a computer science metanotation to bridge the gap between intuitive, informal notations used in reasoning about programming languages, and proof assistants in which facts about the languages can be formally proven.

During the course of the work, I identified a small, well-defined subset of Redex as the input of REDEX-PLUS, and implemented a modular, extensible translation procedure from this Redex subset to a number of proof assistant targets. I have also shown that the generated models can be used to prove theorems about the language models. To my knowledge, REDEX-PLUS is the only tool capable of faithfully translating language models with multiple types of bound variables \(^1\) into multiple proof assistant languages.

\(^1\)For example, term variables and type variables
8.2 Future Work

Generating Binding Lemmas

Many properties of binding operations are useful in proofs. For example, the proof of type preservation of STLC uses a lemma about binding operations (see section 7.2.3). It may be possible to generate a selection of binding lemmas and their proofs as a part of the translation process. LNgen (see section 2.1.1) which generates such lemmas and proofs, may serve as a useful point of reference.

Ellipsis Patterns

Ellipsis patterns, which denote repetitions of sub-patterns, are not supported, not because of any theoretical incompatibilities (which is the reason why, for instance, arbitrary Racket expressions in side-conditions are not allowed), but simply due to complexities and difficulties in implementation. Ellipsis in judgement forms is a similar feature currently unsupported for the same reasons.

Repetition of sub-patterns can be represented as a list containing terms representing these sub-patterns. This would likely require generating data structure definitions to represent the sub-patterns. While lists can be easily defined if not directly importable from the standard libraries, ad hoc list operations and additional constraints on list lengths will be required for more complicated operations (see 3.1.2). However, it is worth noting that even supporting just a small subset of ellipsis patterns (allowing only non-terminals as sub-patterns, for example) will already massively improve the expressiveness of REDEX-PLUS.

Non-terminal Subsets

Overlapping non-terminals are useful for denoting that certain terms (e.g. “values” that cannot be further reduced) are a subset of some other terms (e.g. all expressions). They are problematic to express because REDEX-PLUS’s targets generally do not allow one type (non-terminal) to be the subset of another.

However, it might be possible to side-step this restriction by turning non-terminals denoting a subset of another non-terminal into relations on terms of the other non-terminal. For example, in the case of expressions and values, it might be
possible to represent expressions normally as non-terminals, and represent values as expressions that satisfy a certain condition. Then, all occurrences of values in the language model will be represented by expressions, accompanied by the requirement that it satisfies the “value” condition. This approach will allow ReDEX-PLUS to adequately support many common use cases of overlapping non-terminals, but comes with two major difficulties: first, there needs to be some way of knowing which non-terminals should be represented as a subset of other non-terminals, either manually, or automatically but at the risk of unexpected outcomes; second, inserting the condition might be difficult in places such as function signatures.

Context Closures for Reduction Relations

Context closures are useful for specifying the order of reduction, but are currently unsupported because they require a non-terminal to be used as the reduction context, but defining this non-terminal definition is problematic: it will overlap with other non-terminals, and it will contain holes which are difficult to properly translate. In addition, context closure definitions often use non-terminals that are subsets of other non-terminals (for example, the aforementioned “values” non-terminal), which are not supported either.

To meaningfully support context closures, there needs to be a way to reasonably handle non-terminals intended to be used as reduction contexts. One possible way is to use them only when generating the definition of the context closure, and do not generate code for them at all; alternatively, they can be translated into a predicate that is true only for terms within the reduction context. The generation of the context closure themselves would require first generating the compatible closure, and filtering out cases where the term to be reduced is not in the reduction context.
Bibliography


86


88


Appendix A

Evaluation Proofs

This chapter contains Coq and Agda proofs done for evaluating REDEX-PLUS. See section 7.2 for more details on the proofs.
A.1 Coq Proofs

A.1.1 Progress

Theorem Prog : forall e t, STLC_nil |- e : t -> STLC_Value e \/
  -> exists e', e >>= e'.
Proof with eauto.
  intros e t HT.

  remember STLC_nil as env0.
  generalize dependent Heqenv0.

  induction HT; intros Heqenv0.
  - inversion H; subst.
    + inversion H0.
    + inversion H2.
  - left. autoderive.
  - left. autoderive.
  - right.
    destruct IHHT1; subst...
    + destruct H; inversion HT1; subst.
      exists (sub variable e__a e).
      autoderive.
    + inversion H.
      exists (STLC_app x e__a).
      autoderive.

Qed.

A.1.2 Auxiliary Lemmas for Substitution Lemma

Lemmas about STLC-diff

Lemma same_not_diff : forall v, ~ Is_true (STLC_diff v v).
  Proof with eauto.
intros v.
  unfold STLC_diff.
  rewrite String.eqb_refl..
Qed.

Lemma neq_diff : \forall v v', v \neq v' \leftrightarrow \text{Is_true} (STLC_diff v v').
Proof with eauto.
  intros v v'.
  unfold STLC_diff.
  split.
  - intros Hneq.
    apply String.eqb_neq in Hneq.
    rewrite Hneq.
    unfold Is_true...
  - intros Hd.
    destruct (String.string_dec v v')...
    contradict Hd.
    rewrite e.
    rewrite String.eqb_refl..
Qed.

Main lemmas about changing the environment

Lemma Extend : \forall env env',
  (\forall x t, x : t \leftarrow env \rightarrow x : t \leftarrow env') \rightarrow
  (\forall x y tx ty, x : tx \leftarrow (y : ty :: env) \rightarrow x : tx \leftarrow (y : ty 
\rightarrow :: env')).
Proof using Type with eauto.
  intros env env' HIn x y tx ty HxIn.
  inversion HxIn; autoderive.
Qed.

Lemma Rename : \forall env env',

92
Proof using Type with eauto.
   intros env env' HIn e t HT.
   generalize dependent env'.
   induction HT; intros env' HIn.
   - apply STLCProp_context.
     apply HIn...
   - autoderive.
   - apply STLCProp_abstract.
     apply IHHT.
     intros v t.
     apply Extend...
   - eapply STLCProp_apply...
   Qed.

Corollaries about changing the environment

Corollary Weaken : forall env e t, STLC_nil |- e : t -> env |- e : t.
   Proof with eauto.
   intros env e t HT0.
   apply (Rename STLC_nil env)...
   intros v t0 Hnil.
   inversion Hnil.
   Qed.

Corollary Drop : forall env x e t1 t2 t,
   (x : t1 :: (x : t2 :: env) |- e : t) -> (x : t1 :: env |- e : t).
   Proof with eauto.
   intros env x e t1 t2 t HT.
apply (Rename (x : t1 :: (x : t2 :: env)))...
intros v t0 HIn.
destruct (string_dec x v); subst.
- inversion HIn; subst.
  + autoderive.
  + apply same_not_diff in H5.
    contradiction.
- apply STLCProp_cons.
  + inversion HIn; subst.
    * contradiction.
    * inversion H4; subst...
      contradiction.
  + apply neq_diff...
Qed.

Corollary Swap : forall env x y e t1 t2 t,
  x <> y ->
  (x : t1 :: (y : t2 :: env)) |- e : t ->
  (y : t2 :: (x : t1 :: env)) |- e : t.
Proof with eauto.
  intros env x y e t1 t2 t Hdiff HT.
  apply (Rename (x : t1 :: (y : t2 :: env)))...
  intros v t0 HIn.
  inversion HIn; subst.
  - eapply STLCProp_cons.
    + autoderive.
    + apply neq_diff...
  - inversion H4; subst; autoderive...
Qed.
A.1.3 Substitution Lemma

Lemma Subst :
  \forall env x xt v e t,
  STLC-nil |- v : xt
  \rightarrow x : xt :: env |- e : t
  \rightarrow env |- sub x v e : t.

Proof with eauto.
  intros env x xt v e t HTv HTe.

  generalize dependent x.
  generalize dependent xt.
  generalize dependent env.
  generalize dependent t.

  induction e; intros t env xt HTv x HTe.
  - inversion HTe. subst.
    simpl sub.
    destruct (String.string_dec x s).
    + rewrite <- String.eqb_eq in e.
      rewrite e.
      apply Weaken.
      inversion H1; subst...
      rewrite String.eqb_eq in e.
      subst.
      contradict H6.
      apply same_not_diff.
      + rewrite <- String.eqb_neq in n.
        rewrite n.
        apply STLCProp_context.
        inversion H1; subst...
        rewrite String.eqb_refl in n.
        inversion n.
- inversion HTe.
  simpl sub.
  eapply STLCProp_apply...
- inversion HTe.
  autoderive.
- inversion HTe. subst.
destruct (String.string_dec x s); simpl.
+ rewrite <- String.eqb_eq in e0.
  rewrite e0.
  eapply STLCProp_abstract.
  eapply Drop.
  rewrite String.eqb_eq in e0.
  subst...
+ eapply STLCProp_abstract.
  eapply Swap in H4...
  rewrite <- String.eqb_neq in n.
  rewrite n...
Qed.

A.1.4 Preservation

Theorem Pres : forall e t e', STLC_nil |- e : t --> e --> e' -->
  STLC_nil |- e' : t.
Proof with eauto.
  intros e t e' HT HR.
  remember STLC_nil as env0.
  generalize dependent e'.

  induction HT; intros e' HR;
  inversion HR; subst; try eapply STLCProp_apply...
  eapply Subst...
  inversion HT1...
Qed.
A.2 Agda Proofs

A.2.1 Progress

\[
\text{prog} : \forall e \{t\} \rightarrow \emptyset \vdash e \in t \rightarrow \text{STLC-Value } e \uplus \exists[e'] (e \gg e')
\]

\[
\text{prog} (\text{STLC-v } x) (\text{STLCProp-context } ())
\]

\[
\text{prog} (\text{STLC-app } e_1 e_2) (\text{STLCProp-apply } T_1 T_2) \text{ with prog } e_1 T_1
\]

\[
\ldots | \text{inj}_2 (e_3, e_1R) = \text{inj}_2 (\text{STLC-app } e_3 e_2, \text{STLCProp-beta-cc-app}
\]

\[
\ldots \rightarrow e_1R)
\]

\[
\ldots | \text{inj}_1 x \text{ with e}_1
\]

\[
\ldots | \text{STLC-lam } x_1 x_2 x_3 = \text{inj}_2 ([x_1 \Rightarrow e_2] x_3,
\]

\[
\rightarrow \text{STLCProp-beta-single-step-subst-cc}
\]

\[
\text{prog} (\text{STLC-e-natural } x) V = \text{inj}_1 \text{STLCProp-natural-is-value}
\]

\[
\text{prog} (\text{STLC-lam } x x_1 e) V = \text{inj}_1 \text{STLCProp-lam-is-value}
\]

A.2.2 Auxiliary Lemmas for Substitution Lemma

*Lemmas about STLC-diff*

\[
\text{same-not-diff} : \forall \{x\} \rightarrow \text{STLC-diff } x x \equiv \text{true}
\]

\[
\text{same-not-diff } \{x\} \text{ with } x \neq x
\]

\[
\ldots | \text{no Hneq'} = \bot\text{-elim } (\text{Hneq'} \text{ refl})
\]

\[
\ldots | \text{yes refl with false Data.Bool.Properties.?= true}
\]

\[
\ldots | \text{no Hneq = Hneq}
\]

\[
\text{diff-neq} : \forall \{x x'\} \rightarrow \text{STLC-diff } x x' \equiv \text{true} \rightarrow x \equiv x'
\]

\[
\text{diff-neq } \{x\} \{x'\} \text{ Hdiff with } x \neq x'
\]

\[
\text{diff-neq } \{x\} \{x'\} () \text{ with yes refl}
\]

\[
\ldots | \text{no Hneq = Hneq}
\]

\[
\text{neq-diff} : \forall \{x y\} \rightarrow x \equiv y \rightarrow \text{STLC-diff } x y \equiv \text{true}
\]

\[
\text{neq-diff } \{x\} \{y\} \text{ Hneq with } x \neq y
\]

\[
\ldots | \text{yes refl = } \bot\text{-elim } (\text{Hneq refl})
\]

\[
\ldots | \text{no Hneq' = refl}
\]
Main lemmas about changing the environment

\[
\text{extend} : \forall \{\text{env }\text{ env}'\}
\rightarrow (\forall \{x x' t\} \rightarrow \text{env} \ni x \in t \rightarrow \text{env}' \ni x \in t)
\rightarrow (\forall \{y y' t\} \rightarrow (y \in t :: \text{env}) \ni x \in tx \rightarrow (y \in t :: \text{env}'))
\rightarrow \exists x \in tx)
\]

\[
\text{extend } \text{HIn} \text{ STLCProp-axiom } = \text{STLCProp-axiom}
\]

\[
\text{extend } \text{HIn} \text{ (STLCProp-cons } x \ x_1) = \text{STLCProp-cons } (\text{HIn } x) \ x_1
\]

\[
\text{rename} : \forall \{\text{env }\text{ env}'\}
\rightarrow (\forall \{v v' t\} \rightarrow \text{env} \ni v \in t \rightarrow \text{env}' \ni v \in t)
\rightarrow (\forall \{e e' t\} \rightarrow \text{env} \vdash e \in t \rightarrow \text{env}' \vdash e \in t)
\]

\[
\text{rename } \text{HIn} \text{ (STLCProp-context } x) = \text{STLCProp-context } (\text{HIn } x)
\]

\[
\text{rename } \text{HIn} \text{ STLCProp-natural } = \text{STLCProp-natural}
\]

\[
\text{rename } \text{HIn} \text{ (STLCProp-abstract } HT) = \text{STLCProp-abstract } (\text{rename}
\rightarrow (\text{extend } \text{HIn}) \ HT)
\]

\[
\text{rename } \text{HIn} \text{ (STLCProp-apply } HT HT_1) = \text{STLCProp-apply } (\text{rename } \text{HIn}
\rightarrow HT) (\text{rename } \text{HIn } HT_1)
\]

Corollaries about changing the environment

\[
\text{weaken} : \forall \{\text{env }\text{ e }\text{ t}\} \rightarrow \emptyset \vdash e \in t \rightarrow \text{env} \vdash e \in t
\]

\[
\text{weaken } \{\text{env}\} \ HT = \text{rename } \text{HIn } HT
\]

where

\[
\text{HIn} : \forall \{v t\} \rightarrow \emptyset \ni v \in t \rightarrow \text{env} \ni v \in t
\]

\[
\text{HIn } ()
\]

\[
\text{drop} : \forall \{\text{env }\text{ x }\text{ e }\text{ t}_1 \text{ t}_2 \text{ t}\}
\rightarrow (x \in t_1 :: (x \in t_2 :: \text{env})) \vdash e \in t
\rightarrow (x \in t_1 :: \text{env}) \vdash e \in t
\]

\[
\text{drop } \{\text{env}\} \{x\} \{e\} \{t_1\} \{t_2\} \ HT = \text{rename } \text{HIn } HT
\]

where

\[
\text{HIn} : \forall \{x' x't\} \rightarrow (x \in t_1 :: (x \in t_2 :: \text{env})) \ni x' \in x't
\rightarrow (x \in t_1 :: \text{env}) \ni x' \in x't
\]

98
\[
\text{HIn } \text{STLCProp-axiom} = \text{STLCProp-axiom} \\
\text{HIn } (\text{STLCProp-cons } \text{STLCProp-axiom } x_1) = \bot \text{-elim } (\text{same-not-diff } \rightarrow x_1) \\
\text{HIn } (\text{STLCProp-cons } (\text{STLCProp-cons } \text{HIn' } x) x_1) = \text{STLCProp-cons } \rightarrow \text{HIn' } x
\]

\[
\text{swap} : \forall \{\text{env } x \ y \ tx \ ty \ e \ t\} \\
\rightarrow x \cong y \\
\rightarrow (x \in tx :: (y \in ty :: \text{env})) \vdash e \in t \\
\rightarrow (y \in ty :: (x \in tx :: \text{env})) \vdash e \in t
\]

\[
\text{swap } \{\text{env} \} \{x\} \{y\} \{tx\} \{ty\} \{e\} \{t\} \text{Hneq HT } = \text{rename } \text{HIn HT} \\
\text{where}
\]

\[
\text{HIn} : \forall \{x' \ x't\} \rightarrow (x \in tx :: (y \in ty :: \text{env})) \not\equiv x' \in x't \\
\rightarrow (y \in ty :: (x \in tx :: \text{env})) \not\equiv x' \in x't
\]

\[
\text{HIn } \text{STLCProp-axiom} = \text{STLCProp-cons } \text{STLCProp-axiom } \text{Hneq} \rightarrow \text{Hneq} \\
\text{HIn } (\text{STLCProp-cons } \text{STLCProp-axiom } x_1) = \text{STLCProp-axiom} \\
\text{HIn } (\text{STLCProp-cons } (\text{STLCProp-cons } x x_2) x_1) = \text{STLCProp-cons } \\
\rightarrow (\text{STLCProp-cons } x x_1) x_2
\]

\section*{A.2.3 Substitution Lemma}

\[
\text{subst} : \forall \{\text{env } v \ e \ 	ext{to } vt \ t\} \\
\rightarrow \emptyset \vdash \text{to } \in vt \\
\rightarrow (v \in vt :: \text{env}) \vdash e \in t \\
\rightarrow \text{env } \vdash [v \Rightarrow \text{to }] e \in t
\]

\[
\text{subst } \{v = v\} \text{HTv } (\text{STLCProp-context } \{\text{variable } = x\}) \\
\rightarrow \text{STLCProp-axiom } \text{with } v \neq x \\
\ldots | \text{yes refl } = \text{weaken HTv} \\
\ldots | \text{no Hneq } = \bot \text{-elim } (\text{Hneq refl})
\]

\[
\text{subst } \{v = v\} \text{HTv } (\text{STLCProp-context } \{\text{variable } = x\}) (\text{STLCProp-cons} \\
\rightarrow \text{HIn' Hdiff}) \text{with } v \neq x \\
\ldots | \text{yes refl } = \bot \text{-elim } (\text{diff-neq Hdiff refl})
\]
... | no Hneq = STLCProp-context Hit' 
subst HTv STLCProp-natural = STLCProp-natural 
subst \{v = v\} HTv (STLCProp-abstract \{variable:a = x\} HTe') with v 
\rightarrow \ ?= x 
... | yes refl = STLCProp-abstract (drop HTe') 
... | no Hneq = STLCProp-abstract (subst HTv (swap (\equiv-sym Hneq) 
\rightarrow HTe')) 
subst HTv (STLCProp-apply HTe_1 HTe_2) = STLCProp-apply (subst HTv 
\rightarrow HTe_1) (subst HTv HTe_2)

A.2.4 Preservation

pres : \forall \{e t e'\} 
\rightarrow \emptyset \vdash e \in t 
\rightarrow e \longrightarrow e' 
\rightarrow \emptyset \vdash e' \in t 
pres (STLCProp-context x) ()
pres STLCProp-natural ()
pres (STLCProp-abstract x) ()
pres (STLCProp-apply (STLCProp-abstract x) x_1) (\beta \longrightarrow x_2) = subst 
\rightarrow x_1 x 
pres (STLCProp-apply x x_1) (app \longrightarrow* l y) = STLCProp-apply (pres x 
\rightarrow y) x_1 
pres (STLCProp-apply x x_1) (app \longrightarrow* r x_2 y) = STLCProp-apply x 
\rightarrow (pres x_1 y)