Implementing Certified Programming Language Tools in Dependent Type Theory

by

Adam James Chlipala

B.S. (Carnegie Mellon University) 2003
M.S. (University of California, Berkeley) 2004

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor George C. Necula, Chair
Professor Sanjit A. Seshia
Professor Jack H. Silver

Fall 2007

The dissertation of Adam James Chlipala is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2007

Implementing Certified Programming Language Tools in Dependent Type Theory

Abstract

Implementing Certified Programming Language Tools in Dependent Type Theory

by

Adam James Chlipala

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor George C. Necula, Chair

I present two case studies supporting the assertion that type-based methods enable effective *certified programming*. By certified programming, I mean the development of software with formal, machine-checked total correctness proofs. While the classical formal methods domain is most commonly concerned with after-the-fact verification of programs written in a traditional way, I explore an alternative technique, based on using *dependent types* to integrate correctness proving with programming. I have chosen the Coq proof assistant as the vehicle for these experiments. Throughout this dissertation, I draw attention to features of formal theorem proving tools based on *dependent type theory* that make such tools superior choices for certified programming, compared to their competition.

In the first case study, I present techniques for constructing *certified program verifiers*. I present a Coq toolkit for building foundational memory safety verifiers for x86 machine code. The implementation uses rich *specification types* to mix behavioral require-

ments with the traditional types of functions, and I mix standard programming practice with tactic-based interactive theorem proving to implement programs of these types. I decompose verifier implementations into libraries of components, where each component is implemented as *a functor that transforms a verifier at one level of abstraction into a verifier at a lower level*. I use the toolkit to assemble a verifier for programs that use algebraic datatypes using only several hundred lines of code specific to its type system.

The second case study presents work in *certified compilers*. I focus in particular on *type-preserving compilation*, where source-level type information is preserved through several statically-typed intermediate languages and used at runtime for such purposes as guiding a garbage collector. I suggest a novel approach to mechanizing the semantics of programming languages, based on *dependently-typed abstract syntax* and *denotational semantics*. I use this approach to certify a compiler from simply-typed lambda calculus to an idealized assembly language that interfaces with a garbage collector through tables listing the appropriate root registers for different program points. Significant parts of the proof effort are automated using type-driven heuristics. I also present a generic programming system for automating construction of syntactic helper functions and their correctness proofs, based on an implementation technique called *proof by reflection*.

Professor George C. Necula
Dissertation Committee Chair

Contents

List of Figures **iv**

1 Verification in Context **1**
- 1.1 The State of Formal Software Engineering 1
- 1.2 The Verification Landscape 3
  - 1.2.1 Classical Interactive Theorem Proving 4
  - 1.2.2 Automated Theorem Proving 6
  - 1.2.3 Classical Program Verification with Automated Theorem Proving . . 7
  - 1.2.4 Model Checking 8
  - 1.2.5 Logical Frameworks 9
  - 1.2.6 Theorem Proving and Program Verification via Type Theory 10

2 An Introduction to Coq **15**
- 2.1 The Calculus of Constructions 16
- 2.2 Inductive Types 18
- 2.3 Coq Features and Idioms 21

3 Certified Programming Language Tools **23**
- 3.1 Case Studies in this Thesis 25

I Certified Program Verifiers **30**

4 Programming with Specifications in Coq **31**
- 4.1 Types and Extraction in Coq 31
- 4.2 Modules and Correctness-by-Construction 37
  - 4.2.1 Dependent Types and Composition 38
  - 4.2.2 Modules and Proofs 41

5 A Certified Verifier Toolkit **44**
- 5.1 Introduction 44
  - 5.1.1 Applications to Proof-Carrying Code 45
  - 5.1.2 Contributions 48

5.2 Preliminaries 48
5.2.1 Problem Formulation 48
5.2.2 An Example Input 51
5.3 Components for Writing Certified Verifiers 55
5.3.1 ModelCheck 63
5.3.2 Reduction 75
5.3.3 FixedCode 77
5.3.4 TypeSystem 78
5.3.5 StackTypes 83
5.3.6 SimpleFlags 87
5.3.7 WeakUpdate 90
5.3.8 Architecture Summary 94
5.4 Case Study: A Verifier for Algebraic Datatypes 95
5.5 Implementation 99
5.5.1 The Asm Library 100
5.5.2 Proof Accelerator 101
5.5.3 Certified Verifiers Library 102
5.6 Related Work 103
5.7 Conclusion 106

II Certified Compilers **107**

6 A Language Formalization Methodology **108**
6.1 An Evolutionary Example: The Simply-Typed Lambda Calculus 112
6.1.1 A Nominal Representation 113
6.1.2 De Bruijn Index Representation 114
6.1.3 Higher-Order Abstract Syntax 114
6.1.4 De Bruijn with Dependent Types 121
6.2 Compilation 123
6.3 Related Work 126

7 A Certified Type-Preserving Compiler **127**
7.1 Introduction 127
7.1.1 Task Description 129
7.1.2 Contributions 131
7.1.3 Outline 132
7.2 The Languages 132
7.2.1 Source 133
7.2.2 Linear 135
7.2.3 CPS 137
7.2.4 CC 137
7.2.5 Alloc 138
7.2.6 Flat 143
7.2.7 Asm 144

7.3 Implementation Strategy Overview . . . 147
7.4 Representing Typed Languages . . . 150
7.5 Representing Transformations . . . 152
7.6 Generic Syntactic Functions . . . 154
7.6.1 Generic Correctness Proofs . . . 156
7.7 Representing Logical Relations . . . 157
7.8 Proof Automation . . . 159
7.9 Further Discussion . . . 165
7.9.1 Logical Relations for Closure Conversion . . . 165
7.9.2 Explicating Higher-Order Control Flow . . . 168
7.9.3 Garbage Collection Safety . . . 168
7.9.4 Putting It All Together . . . 169
7.10 Implementation . . . 170
7.11 Related Work . . . 172
7.12 Conclusion . . . 173

8 Generic Programming and Proving 176
8.1 Introduction . . . 176
8.1.1 Outline . . . 179
8.2 Background: Proof by Reflection . . . 180
8.3 Generic Programming and Proving for Simply-Typed Data Structures . . . 182
8.3.1 Reflecting Constructors . . . 183
8.3.2 Reflecting Recursion Principles . . . 185
8.3.3 Generic Proofs . . . 189
8.3.4 A Word on Trusted Code Bases . . . 192
8.4 Managing de Bruijn Indices . . . 193
8.4.1 Dynamic Semantics . . . 194
8.5 A Taste of Manual Implementation . . . 196
8.5.1 Implementing lift . . . 196
8.5.2 Proving liftSound . . . 200
8.5.3 The Prognosis . . . 202
8.6 The Lambda Tamer AutoSyntax System . . . 203
8.6.1 Restricting Denotations . . . 204
8.6.2 Reflecting Denotations . . . 206
8.6.3 Sketch of a Generic liftSound″ Proof . . . 207
8.6.4 Mutually-Inductive AST Types . . . 208
8.6.5 Evaluation . . . 209
8.7 Related Work . . . 210
8.8 Conclusion . . . 212

III Conclusion 214

Bibliography 218

List of Figures

2.1 Syntax of CoC 16
2.2 Typing rules for CoC 16
2.3 Definitional equality rules for CoC 17

5.1 `length.c` 52
5.2 `firstorder.h` 53
5.3 `intlist.h` 53
5.4 A component structure for certified verifiers 56
5.5 Input and output signatures of ModelCheck 62
5.6 Example program for Section 5.3.1 68

7.1 Source language syntax 129
7.2 Target language syntax 129
7.3 Source language dynamic semantics 134
7.4 Syntax of the Linear language 135
7.5 Dynamic semantics of the Linear language 136
7.6 Syntax of the CPS language 137
7.7 Syntax of the CC language 137
7.8 Syntax of the Alloc language 138
7.9 Domains for describing tagged heaps 140
7.10 Dynamic semantics of operands for the Alloc language 141
7.11 Dynamic semantics of terms for the Alloc language 141
7.12 Dynamic semantics of programs for the Alloc language 142
7.13 Syntax of the Flat language 143
7.14 Type language for the first two term languages 149
7.15 Coq definition of syntax and static semantics for Linear 150
7.16 Coq source code of Source syntax and semantics 151
7.17 Continuation composition operator 152
7.18 Linearization translation 152
7.19 Coq source code of the main CPS translation 154
7.20 Coq source code for the first-stage CPS transform's logical relation 159
7.21 Snippets of the Coq proof script for the CPS correctness theorem 166

7.22 Project lines-of-code counts 171

8.1 Simplifying one application of the generic size function 188
8.2 Correct definition of lift″ 199
8.3 Partial initial proof state for UnitIntro case of liftSound″ 201

Acknowledgments

I thank my advisor George Necula, a true hacker who hasn't let tenure rob him of that distinction. George combines a comprehensive knowledge of formal methods with a friendly skepticism about the ability of particular methods to change the world. I've benefited greatly from both his criticism of the over-ambitiousness or impracticality of the ideas I've thought up and from his acceptance of my idealism for those I chose to stick with.

My interest in certified programming sprang from the Open Verifier project, which I worked on with George, Robert Schneck, Evan Chang, and Kun Gao. If Robert hadn't become interested in this whole "interactive theorem proving" business, I can guarantee you that this dissertation would be sporting a different title. Evan has helped me on projects we've collaborated on and with papers I asked him to critique too many times for me to keep count.

I thank "the types people" at Carnegie Mellon University, known at various times as "The Fox Project," "The ConCert Project," or what have you. In particular, I took enlightening and engaging classes on programming languages from Karl Crary, Bob Harper, and Peter Lee, and Bob and his then-student Leaf Petersen served as my guides in my first research experience. It should become apparent how types have invaded this thesis.

I've benefited continuously from being part of the Open Source Quality Project here at Berkeley. I thank everyone who has attended the group lunches for conversations whose tortuous effects on this thesis should not be underestimated.

Part II of this dissertation sprang directly from the POPLmark Challenge. Thanks go to everyone on the `poplmark` electronic mailing list and the attendees of the first Workshop on Mechanizing Metatheory, for making clear the importance of mechanized high-level language semantics and challenging me to defend my ideas against a dogmatic but friendly crowd.

I wouldn't be writing this dissertation if it weren't for my father, who learned programming during a detour from physics and thought it would be a good idea to pass his knowledge on to a six-year-old. I joined the world of "serious programmers" thanks almost entirely to Teen Programmers Unite, one of the earliest Internet-based organizations targeted at and run by young people. I hope that today's new generations manage to overcome the allure of designing flashy web pages and find the same fun in programming that my compatriots and I did at the dawn of the commercial Internet.

I direct a much-deserved thanks to the development team of the Coq proof assistant, for providing such a wonderful playground for a thesis like this one to inhabit.

Finally, thanks to Sanjit Seshia and Jack Silver for their help with this dissertation and for the excursions outside my accustomed world of constructive logic that their graduate classes offered.

Part I of this dissertation is based on a paper [16] presented at ICFP'06, copyright the ACM; and an expanded version currently under revision for publication in the Journal of Functional Programming, copyright Cambridge University Press. Part II is based on a paper [17] presented at PLDI'07, copyright the ACM.

My thesis work was supported in part by a National Defense Science and Engineering Graduate Fellowship, along with National Science Foundation grants CCF-0524784 and CCR-0326577.

Chapter 1

Verification in Context

1.1 The State of Formal Software Engineering

Building software systems that work is notoriously difficult. Though our hardware systems improve dramatically in quantitative performance and capacity from year to year, software seems, if anything, to fail more often. It is often observed that modern computer systems involve more complexity and levels of abstraction than any other human-constructed artifacts in history. The existing body of engineering knowledge has not proved very helpful in taming this complexity, and so the forging of a true discipline of "software engineering" has remained elusive. The prospects seem grim of practicing engineers ever making software reliability guarantees on par with what we expect from other engineering disciplines.

In the 1970's and 1980's, there was a mood of optimism about the power of mathematical methods to solve this problem. In the previous decade, Floyd [36] and Hoare [46] had set out one of the most practically attractive mathematical accounts of program behavior. Not only did their work result in a clear formulation of what a program means, but this meaning was given by a system for building program correctness proofs. That is, what came to be called *Hoare logic* is a tool ready to use in certifying real programs. Building on these ideas, Edsger Dijkstra advocated the technique of *program derivation* [32], where software programs are always developed in concert with their mathematical specifications and correctness proofs. If a programmer starts with an adequate specification for his program, then there is no longer any possibility for bugs to exist at any phase of development, with respect to that specification.

Unfortunately, upon returning our attention to the present day, we see almost zero adoption of program derivation, or even of formal program correctness techniques of any kind. What went wrong? Computing has only become *more* pervasive, creating greater need to know that our computer systems do their jobs. Yet somehow the best intentions and efforts of a generation of computer scientists failed to yield practical tools and their associated methodology for building certified correct software.

At this point, I want to suggest that such a conclusion is drawn too hastily. We should take another look at the state of the art today, this time not neglecting an important kind of "stealth formalism" that has become absolutely standard in software development. This is *static type systems*. In mainstream programming languages, we have become dulled to the novelty of writing proofs (type annotations) inline in our program source code and requiring that our programs satisfy mechanized proof checkers (type checkers) before we would even think about running them. The theorems we prove are of a particular mundane kind, often called by the names "type safety" and "memory safety," but they are non-trivial theorems nonetheless, as would probably be obvious to a programmer in the early days of assembly language. While most formal methods advocates today find memory safety theorems dull, these theorems have an important role to play: They make it possible to reason about programs *modularly*. Since non-memory-safe programs may overwrite segments of memory arbitrarily, one buggy component may break the key invariants of all the other components with which it coexists. Thus, whether the reasoning to be applied is formal or informal, the ability to exercise it strikes an important blow in the battle against software system complexity.

The techniques of type systems have succeeded in producing usable systems for certifying memory safety with programmer cooperation. Can we learn from what has worked in "the real world" (type systems) and what hasn't (Hoare logic and classical program verification) to bring ourselves closer to the 1970's vision of full verification? **In this thesis, I argue that richer type systems enable practical total correctness verification of real systems, and I demonstrate this through two significant certified programming projects performed with a type-based computer proof assistant.**

In the next section, I will briefly survey which sorts of tools are available today. I will critique the approaches and motivate why I chose a particular tool, the Coq proof assistant [8], for this thesis work.

1.2 The Verification Landscape

Formal verification has been a part of computer science from the field's early days. As such, it's not surprising that many different tools have sprung up with different strengths and weaknesses. As the title of this dissertation may have given away, I will be describing experiments in total correctness verification of programming languages tools, such as compilers and program analyzers. In this section, I will survey the verification approaches that exist today, with language tool verification in mind. I will be able to dismiss some techniques outright as not equal to such a complex task, and others will remain, leading up to a final weighing of their pros and cons and a decision of which tool to use.

1.2.1 Classical Interactive Theorem Proving

One of the earliest interactive theorem proving tools, or "proof assistants," was Milner's LCF system [61]. It was described as a "smart blackboard," where operators could perform mathematical proofs in a way that precluded invalid leaps of reasoning. The concrete interaction model was a loop alternating between the operator entering a *tactic*, a short command describing a proof step; and the system updating a display summarizing the state of the proof effort. Each tactic was checked by the proof assistant for logical validity before being used to update proof state.

Tactic-based theorem proving has endured to the present day, with many academic users but no appreciable amount of industrial adoption. A confusing profusion of proof assistants exists today. However, they can mostly be divided into two broad categories: the "LCF-style" provers, where a proof is a sequence of tactics; and type-theoretical provers, where a proof is a term of a typed lambda calculus, usually *constructed* with the aid of tactics. Type-theoretical provers are the subject of Section 1.2.6, and I elaborate on LCF-style provers in rest of this subsection.

The most common LCF-style provers today are Isabelle [73], a direct descendant retaining the historical association of the LCF approach with the ML family of programming languages; and PVS [70], a Lisp-based proof assistant developed at SRI. Isabelle is a framework for supporting new proof systems, though it is most often used in its "Isabelle/HOL" guise, which uses a classical higher-order logic HOL [38]. The dedicated user communities of Isabelle/HOL and PVS may grumble at my lumping them together, as HOL features a much smaller and simpler set of primitive tactics than PVS does, but they are more similar than different compared to any of the alternatives I discuss later.

The LCF approach is no doubt the most widely used among practical interactive theorem proving efforts today. This is probably due to the ease of extending a proof assistant with new reasoning tricks: one simply adds new proof rules that embody them. The proof assistant developer can certainly make soundness mistakes in implementing new rules, but, if he doesn't, users can rest assured that they aren't producing bogus proofs. As a result, Isabelle/HOL and PVS are remarkably effective out-of-the-box at automating large parts of proofs, while still providing good support for solicitation of human insight.

However, there is a dark side to this convenience. To trust in the soundness of an LCF-style prover, you must trust in the correct implementation of each of its primitive proof rules. If one of these rules implements the simplex method for deciding linear arithmetic formulas, then you need to trust that the simplex algorithm was implemented correctly. In today's networked world of computing, we prefer to minimize the amount of code that we need to trust to obtain suitable security guarantees. In applications like proof-carrying code [67], which is the setting for Part I of this dissertation, any bug in proof-checking machinery opens opportunities for attackers to run malicious code.

There are also many benefits associated with type-theoretical representation of proofs that are lost in the LCF style. We will meet these benefits throughout this dissertation, and I sketch their major elements in Section 3.1. The main idea can be summarized as: **By choosing a simple proof representation embeddable in a statically-typed programming language, we gain the ability to verify our proof-generating and -manipulating programs by type-checking them.**

1.2.2 Automated Theorem Proving

A competing theorem proving tradition is that of automated provers, perhaps most widely associated with Nqthm, or "the Boyer-Moore prover" [10]. The system was designed for the verification of Lisp programs, but it and its modern successor ACL2 [50] have come to be used as general-purpose theorem proving tools, with applicative Common Lisp serving as a modeling language for wider domains.

These tools have been used to accomplish some impressive certifications, including the verification of a stack from a high-level programming language to a machine architecture [63]. Nonetheless, the drawbacks I listed for LCF-style provers apply only more strongly here. Since proofs are expected to be much more automatic, there are only *more* fundamental reasoning steps whose implementations must be trusted, and the lack of wide industrial adoption of ACL2, a quite dynamically-typed system, opens the door to speculation on how types may be put to good use here.

1.2.3 Classical Program Verification with Automated Theorem Proving

The most successful family of tools implementing classical, Hoare logic-style verification has been the Extended Static Checking (ESC) family [35]. Implemented for such languages as Modula-3 and Java, the technique depends on programmers annotating their functions with preconditions and postconditions and their loops with invariants. This burden is light compared to interactive theorem proving, as an automated, first-order theorem prover is called to fill in the blanks. The traditional theorem prover of choice has been Simplify [30], developed by the same group of authors and evolved in parallel. It uses the Nelson-Oppen architecture [69] for cooperating decision procedures, which was designed specifically to handle the kinds of proof obligations arising from ESC-style verification.

This approach has been successful at what I'll call "laundry list" verification, where one wants to check that a program satisfies a list of standard, generic properties, such as absence of out-of-bounds array accesses. Effective automated proving for the theories of equality, linear arithmetic, and arrays supports such verifications. However, ESC has not been shown to scale to total correctness proofs for such complicated programs as compilers. It seems believable that a human user is going to have to write *something* that looks like a proof to facilitate such an in-depth verification, where the necessary structure of the proof is more complicated than the structure of the underlying program. The ESC approach in essence makes exactly the opposite assumption, requiring that a program is proved correct by an inductive argument mirroring its control structure, with annotations providing induction hypotheses. As I am interested in this thesis in verifying symbolic programs with subtle correctness arguments, I will abandon the ESC approach here.

1.2.4 Model Checking

A very different approach that first appeared in the early 1980's is model checking [18, 80], which traditionally has been centered around temporal logic specifications and verification of finite-state systems by exhaustive enumeration. Brute force search is obvious enough when designing algorithms that need only terminate in theory, but many innovations have been necessary to make such search practical on real machines, with milestones including the use of ordered binary decision diagrams in "symbolic model checking" [11].

More recently, the early 2000's saw the expansion of model checking for software beyond finite-state programs, through integration with automated first-order theorem provers, especially using the techniques of predicate abstraction and counterexample-guided abstraction refinement [5, 43]. While this kind of tool has traditionally only been effective for verification of safety properties describable by finite-state monitors over sequential programs, recent work has branched into concurrency [42], liveness properties [19], and heap shape analysis [26].

Model checking has made many important inroads, and it is probably the most used formal verification technique outside of academia. However, for the total correctness verification of complicated symbolic programs, the situation is even worse than for ESC, as model checking generally assumes lack of programmer annotations. Exhaustive exploration of abstracted state spaces can be one useful decision procedure among many in interactive verification, but few would predict that a model checker will ever perform a total correctness verification of a compiler.

1.2.5 Logical Frameworks

The idea of "logical frameworks" has been proposed for distilling the essential features required for formalization of programming languages, logics, and their metatheories. The Edinburgh Logical Framework (LF) [41] is the most well-known of these, and its most popular tool implementation today is Twelf [76]. LF provides a tiny dependently-typed lambda calculus that is parameterized on a *signature* that lists, for instance, the logical constants used to describe the syntax and semantics of a programming language. Some of these constants are types, while others inhabit those types. Twelf provides a logic programming interpretation of signatures that can be used to deduce that certain types represent total logic programs. These logic programs can, in turn, be treated as "meta-theorems" about the programming language or logic that has been formalized, otherwise known as the *object language*, in contrast to the *meta language* LF.

LF is a simple enough language that it supports the idea of *higher-order abstract syntax (HOAS)* [75] soundly, in contrast to the languages we will meet in the next subsection. As Part II will highlight, dealing with *variable binding* is one of the most pervasive challenges in proving theorems about high-level programming languages, and HOAS makes many of the important theorems therein implicit in the meta language.

However, Twelf provides no significant proof automation support; users must write out every detail of every proof. Part of this can be attributed to social accident, with more engineering effort having been put into proof assistants proper than into Twelf. There are also significant Twelf design decisions that make automation comparable to that found in LCF-style provers unlikely. Twelf's logic program "meta-proofs" are not first-class objects that may be passed around. Rather, the user writes out a signature and then asks Twelf to check it for totality. Within Twelf, it's not possible to write anything like a program that generates a meta-proof (as tactic languages allow), let alone verify such programs statically (as is possible in the tools we meet in the next subsection). For these reasons, as well as Twelf's historical lack of support for program verification, I abandon it here.

1.2.6 Theorem Proving and Program Verification via Type Theory

The broad approaches surveyed so far in this section take the (intuitively reasonable) tack of labeling programming and theorem proving as two distinct tasks, motivating the development of separate mechanisms in support of each. An alternative philosophy is based on the so-called "Curry-Howard correspondence" [86], an idea quite popular in the functional programming community. The original correspondence provided a precise connection between propositional constructive logic and a simply-typed lambda calculus. *Constructive logic* contrasts with classical logic, the almost universal formal foundation of mathematics outside of isolated areas of mathematics and computer science, by *defining truth as provability*. Common surprises upon moving from classical to constructive logic include the invalidity of the law of the excluded middle and the inability to prove existentially quantified facts without demonstrating *concrete instantiations* for the quantifiers. In classical logic, you might be able to prove that there exists a program that performs a certain function, but still be unable to present any such concrete program. In constructive logic, you need not worry about such situations; from any proof, you can read off the program that exists via a mechanical procedure.

This kind of property makes the correspondence possible. Concretely, Curry-

Howard makes the surprising claim that every *theorem statement* of the logic can be interpreted as a *type* of the lambda calculus, and vice versa; and that every *proof* of the logic can be interpreted as a *program* of the lambda calculus, and vice versa.

The connection becomes obvious in retrospect when we set some constructive logic proof rules and lambda calculus typing rules side by side. For instance, we can consider the rules governing *implication* and *function types.*

$$\frac{\Gamma, \phi_1 \vdash \phi_2}{\Gamma \vdash \phi_1 \supset \phi_2} \supset\mathrm{I} \quad \frac{\Gamma \vdash \phi_1 \supset \phi_2 \quad \Gamma \vdash \phi_1}{\Gamma \vdash \phi_2} \supset\mathrm{E}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 .\ e : \tau_1 \to \tau_2} \to\mathrm{I} \quad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \to\mathrm{E}$$

We use a standard natural deduction proof theory for constructive logic, where the primary way of deducing a formula built from a connective is to use that connective's *introduction rule*, and a connective is *used* through its *elimination rule*. In programming, these have their counterparts in program constructs thought of as *constructors* and *destructors*, respectively. In fact, the Curry-Howard correspondence is so deeply entrenched in today's functional programming community that the terminology "introduction and elimination rules" is used more commonly, and so I label both kinds of rules with names ending in I and E to signify their sorts.

There are two important differences between the two very similar pairs of rules. First, the proof judgment $\Gamma \vdash \phi$ merely declares that the truth of formula ϕ follows from the truth of the formulas in the list, or *context*, Γ. In contrast, the program typing judgment

$\Gamma \vdash e : \tau$ does more than just assert that there exists a program of type τ with free variables whose types are given by context Γ; the judgment also provides a concrete program e of that type. In the context of program type-checking, it's obvious that we need to think about concrete programs, but it's less obvious that transplanting the idea to the logical setting suggests a concrete proof format. We can **use typed lambda calculus programs as a concrete format for writing *proof terms* that witness the constructive validity of formulas**.

The other important difference is that, in the natural deduction proof system, we can simply add formulas to what are effectively sets of known formulas. These formulas in the context are thought of as *hypotheses* in proofs with nested hypothetical structure. Any hypothesis establishing a formula is as good as any other, so there's no reason to assign hypotheses names in rule $\supset$I. In contrast, in programming, we're quite used to the idea that one integer needn't be as good as any other, so we assign bound variables names in rule $\rightarrow$I. In using lambda calculus terms as proofs, it is also useful to name hypotheses, to make proof checking programs easier to write.

It turns out that this exercise can be continued for all of the usual logical connectives and their type system counterparts. We arrive at a convenient computer format for storing constructive proofs, with the foundations of programming language semantics available to guide us in the various design decisions. Though this is already a useful trick, we can push even further and make the surprising decision to *unify programming and proving completely*. Since each program type matches up with a logical connective and vice versa, we will mix them freely. Sometimes we think of ourselves as programming, sometimes as proving, and sometimes as *doing both at once*. We can *program with specifications and proofs*, enhancing traditional programs with types that express arbitrary correctness specifications. To call such a function, pass it a proof that its "precondition" holds; and you can expect it to return a proof of its "postcondition" along with the traditional result. This result itself can involve nested uses of rich types.

A good number of proof assistants have been implemented along these lines, including Agda [20], Coq [31], and Lego [33]. Through the Curry-Howard lens, each can be thought of as not just a formal proof construction environment, but also a development environment for an expressively-typed functional programming language. As rich types can make program construction as difficult as proof construction, the theorem proving tools can come in handy even in tasks that seem best classified as "programming."

I chose to use Coq for the projects described in this dissertation, as it has the most mature type-theoretical proof assistant implementation. Many useful tools have been developed in the Coq ecosystem and eventually integrated into the main distribution. This includes excellent support for the development of certified programs via a procedure for compiling Coq terms to native code through a partially-type-erasing translation to OCaml. It is possible to develop in Coq *proof-manipulating programs* whose types guarantee that they always produce correct proofs when given correct proofs as inputs. That idiom is at the core of this thesis, and I will argue that it combines the strengths of classical type-based programming and classical theorem proving, when used with a robust proof assistant like Coq.

Before beginning to describe my contributions in detail, I will spend Chapter 2 on an introduction to the pertinent aspects of Coq's mathematical foundations and usage. Chapter 3 introduces the domain of certified programming language tools, motivates their utility as case studies in certified programming, and summarizes the cases studies that make up my thesis. Parts I and II present these case studies in detail: a certified machine code analysis system and a certified compiler for a functional programming language.

Chapter 2

An Introduction to Coq

The Coq proof assistant [8] is a type-theoretical theorem proving and certified programming tool based on an underlying logic/programming language, the *Calculus of Inductive Constructions (CIC)*. CIC is a *logic* because it can be used to state approximately all theorems and express approximately all proofs that show up in mathematics.[1] CIC is a *programming language* because it is a typed lambda calculus that can be used to write *certified programs*, using the language's mathematical capabilities to express strong specifications through typing.

In this chapter, I give an overview of CIC, starting with its simpler subset, the Calculus of Constructions (CoC) [21], in Section 2.1. CIC builds on CoC by adding a primitive *inductive type definition* facility, which I describe less formally in Section 2.2.

[1] I write "approximately" not to indicate formal incompleteness, but rather to acknowledge that there are many choices to be made in the foundations of mathematics, and CIC hard-codes some of these choices. In practice, this only affects esoteric results that don't bear directly on practical engineering problems, which are my focus in this thesis.

$$\begin{array}{rrcl} \text{Variables} & x & & \\ \text{Naturals} & n & \in & \mathbb{N} \\ \text{Terms} & E & ::= & x \mid E\ E \mid \lambda x : E.\ E \mid \forall x : E.\ E \mid \mathsf{Type}_n \end{array}$$

Figure 2.1: Syntax of CoC

$$\frac{\Gamma(x) = E}{\Gamma \vdash x : E} \quad \frac{\Gamma \vdash E_1 : (\forall x : E_{\mathsf{dom}}.\ E_{\mathsf{ran}}) \quad \Gamma \vdash E_2 : E_{\mathsf{dom}}}{\Gamma \vdash E_1\ E_2 : E_{\mathsf{ran}}[x \mapsto E_2]}$$

$$\frac{\Gamma \vdash E_{\mathsf{dom}} : \mathsf{Type}_n \quad \Gamma, x : E_{\mathsf{dom}} \vdash E : E_{\mathsf{ran}}}{\Gamma \vdash (\lambda x : E_{\mathsf{dom}}.\ E) : (\forall x : E_{\mathsf{dom}}.\ E_{\mathsf{ran}})}$$

$$\frac{\Gamma \vdash E_{\mathsf{dom}} : \mathsf{Type}_n \quad \Gamma, x : E_{\mathsf{dom}} \vdash E : \mathsf{Type}_n}{\Gamma \vdash (\forall x : E_{\mathsf{dom}}.\ E_{\mathsf{ran}}) : \mathsf{Type}_n}$$

$$\frac{}{\Gamma \vdash \mathsf{Type}_n : \mathsf{Type}_{n+1}} \quad \frac{\Gamma \vdash E : \mathsf{Type}_n}{\Gamma \vdash E : \mathsf{Type}_{n+1}}$$

$$\frac{\Gamma \vdash E_1 : E_2' \quad E_2' \equiv E_2}{\Gamma \vdash E_1 : E_2}$$

Figure 2.2: Typing rules for CoC

2.1 The Calculus of Constructions

Figure 2.1 presents the abstract syntax of CoC. We can recognize the elements of the simply-typed lambda calculus and System F [77], but CoC has an important difference from these systems. "Terms" and "types" are collapsed into a single syntactic class. Thus, alongside variables and function abstraction and application, we find the function type form $\forall x : E_1.\ E_2$, denoting a function with domain E_1 and range E_2. This is a *dependent function type* because the variable x is bound in E_2 to the *value* of the function argument. I write $E_1 \to E_2$ as a shorthand for $\forall x : E_1.\ E_2$ when x is not free in E_2.

Figure 2.2 gives the complete set of typing rules for CoC. The most interesting part

$$\frac{}{(\lambda x : E_{\mathsf{dom}}.\ E_1)\ E_2 \equiv E_1[x \mapsto E_2]}$$

$$\frac{E_1 \equiv E_1' \quad E_2 \equiv E_2'}{E_1\ E_2 \equiv E_1'\ E_2'} \quad \frac{E_1 \equiv E_1' \quad E_2 \equiv E_2'}{\lambda x : E_1.\ E_2 \equiv \lambda x : E_1'.\ E_2'}$$

$$\frac{E_1 \equiv E_1' \quad E_2 \equiv E_2'}{\forall x : E_1.\ E_2 \equiv \forall x : E_1'.\ E_2'}$$

$$\frac{}{E \equiv E} \quad \frac{E' \equiv E}{E \equiv E'} \quad \frac{E \equiv E' \quad E' \equiv E''}{E \equiv E''}$$

Figure 2.3: Definitional equality rules for CoC

of Figure 2.2 is the last rule, which says that if two terms are *definitionally equal*, then, when considered as types, they describe the same set of terms. Figure 2.3 presents the definitional equality relation $\equiv$, an equivalence relation that models usual computational reduction rules. A crucial property of this definition of $\equiv$ is that it is decidable when restricted to well-typed terms. Here we see a standard property of dependently-typed languages like CoC: each of the static and dynamic semantics depends on the other in a key way.

We also have terms Type_n denoting different levels in an infinite type hierarchy. Figure 2 shows that any Type_n has type Type_m for any $m > n$. That is, types themselves are first-class entities that are classified by other types. Because of this, we don't need separate constructs for normal functions and parametric polymorphism, as in System F. Rather, we can write System F-style terms using CoC restricted to using only type level 0 in our definitions. For instance, $\lambda\tau : \mathsf{Type}_0.\ \lambda x : \tau.\ x$ is the polymorphic identity function and has type $\forall\tau : \mathsf{Type}_0.\ \tau \to \tau$.

Readers accustomed to System F may be growing irritated at the necessity to calculate the right Type indices to include in various places. System F supports *impredicative* type quantification, where quantified types may be instantiated with themselves. Type indices in CoC as presented here forbid such circularities. There is an impredicative variant of CIC that may be turned on in Coq by way of a command-line flag, and, in fact, even the standard language supports a special impredicative sort for logical propositions. I stick to the completely *predicative* version here for simplicity of exposition.

2.2 Inductive Types

If we are just concerned with "programming" of the traditional kind, then CoC is quite sufficient for encoding a wide variety of features, following the standard trick of "encoding types with their polymorphic elimination forms." However, we run into trouble when we attempt to use the same system to encode formal proofs about these programs, let alone proof-manipulating, dependently-typed programs of the kinds that will appear later. Basic properties of data structures turn out unprovable, and we encounter algorithmic inefficiencies with encoding schemes that try to fit so many features into so simple a base language. Additionally, it sometimes turns out that the natural encodings of data types are inhabited by more terms than we expected them to be. For these reasons, Coq was enriched with special support for what are called *inductive types* [72], a very general type definition mechanism that can be used to define all of the standard data types and logical connectives. The enhanced logic is called the Calculus of Inductive Constructions (CIC).

Rather than provide an exhaustive formalization, I will demonstrate inductive types by example, relating the examples to the formalization of CoC.

Inductive types subsume the algebraic datatypes found in subsets of Haskell and

ML that disallow infinite data structures and non-terminating programs, as demonstrated by this definition of the natural numbers:

$$\begin{array}{l}\mathsf{Inductive\ nat} : \mathsf{Type}_0 := \\ \quad \mid \mathsf{O} : \mathsf{nat} \\ \quad \mid \mathsf{S} : \mathsf{nat} \rightarrow \mathsf{nat}\end{array}$$

We define a type named nat at type level 0 by describing how its values may be built. Its two *constructors* are O, the natural number 0; and S, the function from a natural number to its successor. Thus, the natural numbers are denoted O, S O, S (S O), etc..

We can perform pattern matching on nats:

$$\begin{array}{rcl}\mathsf{pred} & = & \lambda x : \mathsf{nat}.\ \mathsf{match}\ x\ \mathsf{with} \\ & & \quad \mid \mathsf{O} \Rightarrow \mathsf{O} \\ & & \quad \mid \mathsf{S}\ n \Rightarrow n\end{array}$$

...and we can write recursive functions over them:

$$\begin{array}{rcl}\mathsf{plus} & = & \mathsf{fix}\ f\ (n : \mathsf{nat}) : \mathsf{nat} \rightarrow \mathsf{nat}.\ \mathsf{match}\ n\ \mathsf{with} \\ & & \quad \mid \mathsf{O} \Rightarrow \lambda n' : \mathsf{nat}.\ n' \\ & & \quad \mid \mathsf{S}\ n \Rightarrow \lambda n' : \mathsf{nat}.\ \mathsf{S}\ (f\ n\ n')\end{array}$$

Coq enforces that every recursive call in one of these fix expressions passes a recursive argument that is a syntactic subterm of the current argument. This retains the important property of CoC that every program "terminates" in a suitable sense, allowing us to keep the relation $\equiv$ decidable and preclude such tricks as presenting an infinite loop as a "proof" of any proposition.

We can also define parameterized inductive types, such as lists:

$$\begin{array}{l}\mathsf{Inductive\ list}\ (T : \mathsf{Type}_0) : \mathsf{Type}_0 := \\ \quad \mid \mathsf{nil} : \mathsf{list}\ T \\ \quad \mid \mathsf{cons} : T \rightarrow \mathsf{list}\ T \rightarrow \mathsf{list}\ T\end{array}$$

...and write recursive functions over them:

$$\begin{array}{rcl} \mathsf{length} & = & \mathsf{fix}\ f\ (T : \mathsf{Type}_0)\ (\ell : \mathsf{list}\ T) : \mathsf{nat}.\ \mathsf{match}\ \ell\ \mathsf{with} \\ & & \quad \mid \mathsf{nil} \Rightarrow O \\ & & \quad \mid \mathsf{cons}\ \_\ \ell' \Rightarrow \mathsf{S}\ (f\ T\ \ell') \end{array}$$

Inductive types also generalize the *generalized algebraic datatypes (GADTs)* [81] that have recently crept into reasonably wide use through a popular extension to Haskell. For example, consider this definition of a list-like type family indexed by natural numbers providing strict bounds on list lengths.

$$\begin{array}{l} \mathsf{Inductive}\ \mathsf{listN}\ (T : \mathsf{Type}_0) : \mathsf{nat} \to \mathsf{Type}_0 := \\ \quad \mid \mathsf{nilN} : \mathsf{listN}\ T\ \mathsf{O} \\ \quad \mid \mathsf{consN} : \forall n : \mathsf{nat}.\ T \to \mathsf{listN}\ T\ n \to \mathsf{listN}\ T\ (\mathsf{S}\ n) \end{array}$$

The only value of $\mathsf{listN}\ T\ \mathsf{O}$ is the nil list, every value of $\mathsf{listN}\ T\ (\mathsf{S}\ \mathsf{O})$ is a one-element list, etc.. In contrast to the case of GADTs, where all parameters of the type being defined must be types, inductive types allow arbitrary "term-level" data to appear in type indices. This lets us use the types of functions to be much more specific about their behavior than in traditional programming:

$$\begin{array}{rcl} \mathsf{countdown} & = & \mathsf{fix}\ f\ (n : \mathsf{nat}) : \mathsf{listN}\ \mathsf{nat}\ n.\ \mathsf{match}\ n\ \mathsf{with} \\ & & \quad \mid \mathsf{O} \Rightarrow \mathsf{nilN}\ \mathsf{nat} \\ & & \quad \mid \mathsf{S}\ n' \Rightarrow \mathsf{consN}\ \mathsf{nat}\ n'\ n\ (f\ n') \end{array}$$

Actual Coq code must be a bit more verbose about this, with extra type annotations to make type inference tractable. However, I will omit those details here in the interest of clarity.

We can also take advantage of the possibility to create data structures that contain types. Here's an example of defining the type of lists of types, followed by the type family of tuples whose component types are read off from particular type lists.

$$\begin{array}{l}
\mathsf{Inductive\ listT} : \mathsf{Type}_1 := \\
\quad \mid \mathsf{nilT} : \mathsf{listT} \\
\quad \mid \mathsf{consT} : \mathsf{Type}_0 \to \mathsf{listT} \to \mathsf{listT} \\
\\
\mathsf{Inductive\ tuple} : \mathsf{listT} \to \mathsf{Type}_2 := \\
\quad \mid \mathsf{Nil} : \mathsf{tuple\ nilT} \\
\quad \mid \mathsf{Cons} : \forall T : \mathsf{Type}_0.\ \forall \ell : \mathsf{listT}. \\
\qquad T \to \mathsf{tuple}\ \ell \to \mathsf{tuple}\ (\mathsf{consT}\ T\ \ell)
\end{array}$$

As an example, one value with type $\mathsf{tuple}\ (\mathsf{consT}\ \mathsf{nat}\ (\mathsf{consT}\ (\mathsf{nat} \to \mathsf{nat})\ \mathsf{nilT}))$ is $\mathsf{Cons}\ \mathsf{nat}\ (\mathsf{consT}\ (\mathsf{nat} \to \mathsf{nat})\ \mathsf{nilT})\ O\ (\mathsf{Cons}\ (\mathsf{nat} \to \mathsf{nat})\ \mathsf{nilT}\ (\lambda x : \mathsf{nat}.\ x)\ \mathsf{Nil})$. Coq supports an *implicit argument* facility that lets us write this term as $\mathsf{Cons}\ O\ (\mathsf{Cons}\ (\lambda x : \mathsf{nat}.\ x)\ \mathsf{Nil})$. Though I won't explain the details of implicit arguments here, I will trust in the reader's intuition as I freely omit inferable arguments in the following discussion.

Note that both inductive type families that I define here exist at type levels above 0. The rule in effect here is that an inductive type must live at at least type level $n + 1$ if one of its constructors' types itself has type Type_{n+1} but not type Type_n. The intuition is that inductive definitions end up at higher type levels as they employ more sophisticated uses of "types as data." In actual Coq code, Type indices are inferred automatically, and the user only needs to worry about them when the Coq system determines that no feasible solution exists to an induced set of constraints on index variables. Following that discipline, I will use Type unadorned in the remainder of this dissertation.

2.3 Coq Features and Idioms

Rather than attempt to lay out all Coq background material in this chapter, I've opted to defer the discussion of each further topic until directly before it is used in the following chapters. In summarizing my case studies, Section 3.1 provides an index for which chapters describe which Coq idioms, including tactic-based proving, specification types, modules, and proof by reflection.

Chapter 3

Certified Programming Language Tools

I began this work with the motivation of optimizing extensible proof-carrying code [67] architectures. Part I provides more background on that topic, but, by the time I'd completed the work described in this dissertation, my focus had shifted to discovering techniques for effective construction of certified, dependently-typed programs. This new problem is closer to the center of the software engineering crisis that is on the minds of many researchers in programming languages and elsewhere today. Serendipitously enough, it turns out that programming language tools tend to share a number of features that make them particularly good candidates for formal certification case studies.

First, certifying tools that manipulate code provides a foundation for certifying applications. By proving that compilers preserve semantics, we can prove theorems about source code instead of machine code, without sacrificing guarantees about the programs that we *really* run. By proving that machine code analyzers enforce memory safety, we are free to use modular reasoning about those programs, without needing to worry that a buffer overrun somewhere ignores well-specified interfaces.

Second, there are well-established, rigorous standards of what it means for various programming language tools to be correct. We have the standard methodologies of abstract interpretation [23] and structured operational semantics [77], for instance. The promise of formal verification is often challenged with the canonical "But how do you write a specification for Microsoft Word?" objection, but we have no such trouble specifying compilers and analysis tools, some of formal semanticists' favorite examples since the early days.

Finally, programming language tools are frequently written in a purely functional style with no non-terminating functions. We've seen that it is important that Coq forces termination, since proofs are intermixed freely with programs, and a non-terminating proof generator ought not to be treated as a universal prover. While Coq enforces this by forcing all recursive function definitions to follow very regular recursion schemes, almost all of the kinds of functions we associate with programming language tools tend to be written via simple recursive case analysis in the first place. A common exception is a main loop that explores a state space or calculates a fixed point, but it is easy to express these as recursive in a new "time-out" parameter, which counts how many more steps we are willing to allow. In practice, we can start these processes with very large time-outs and notice no difference from unbounded implementations, and the bothersome reasoning about time-outs is confined to the final steps of correctness proofs.

3.1 Case Studies in this Thesis

The bulk of the rest of this dissertation is divided into two parts, describing two concrete certified tool implementations. Each of these can be read as something like a fable, where the moral at the end of the story is that a particular set of idioms for using dependent types can play a critical role in making certified programming practical.

Part I deals with *certified program verifiers*; specifically, machine code memory safety verifiers useful in the context of extensible proof-carrying code systems. We want to know that programs downloaded from an untrusted source are memory safe before running them. We could fix a particular verification strategy, as used in bytecode verification approaches, but then we could only validate programs prepared with that narrow verification algorithm in mind. When the next big thing in programming languages arrives with its fancy new compilation strategy, the existing fixed verifier will most likely be unable to cope. Instead, we'd like to allow language implementors to distribute their own customized verifiers. To allow us to trust these verifiers' results, we require that each verifier come with a *proof of correctness*.

This idea is simple enough to describe, and, in theory, it helps us reach a sweet spot between verification flexibility and run-time efficiency. Standing in the way of a practical deployment are the software engineering and proof engineering issues of building such things. In Part I, I describe the implementation of a toolkit for modular construction of certified verifiers in Coq. Chapter 4 introduces the two main idioms that make this feasible.

First, we have *specification types*, including the *subset types* from the Coq standard library. A subset type pairs a base type with a predicate over that type. The new type is inhabited by pairs of base values and proofs that they satisfy the predicate. I extend the repertoire of specification-oriented types with alternatives suitable for representing the results of *incomplete decision procedures* and incomplete functions in general, which pop up unavoidably in writing tools that compute approximate solutions to undecidable verification problems. I demonstrate how Coq's macro facility and tactic library can be put to good use in enabling a programming style based on treating specification type families as *failure monads*, helping to alleviate the burden associated with threading proofs through a certified program.

Second, we have Coq's ML-style module system. By working with *functors that manipulate proofs as well as programs*, we can construct a library of components for building certified verifiers. I adopt a particular novel decomposition of the problem. From the compilers literature, we're familiar with compiling from high-level languages to low-level languages through a succession of intermediate languages. In my verifier toolkit, I provide the tools to "decompile" a machine code program to the level where a simple set of typing rules can be used to validate it. In fact, the program stays the same; what really evolves is *what abstraction we use to analyze it.* The lowest level abstraction models a real machine architecture, while the progression to higher level abstractions adds more and more state that we assume is provided by an "oracle." At every stage, correctness proofs are provided that are parametric in the proof obligations of higher-level abstractions yet to be chosen.

Chapter 5 uses these ideas to complete a concrete case study within a case study, a memory safety verifier for x86 machine code programs compiled from a fictitious language that supports algebraic datatypes.

Part II deals with *certified compilers*, a more well-known subject. Proofs that compilers preserve program behavior have been part of computer science lore for decades. All of the past projects that I know of end with proclamations that some compiler has been proved correct, without much attention to how the proof engineering process can be streamlined. It seems fair to say that a non-researcher looking to certify his compiler today has no guidance or tool support strong enough to make certification less than a PhD thesis worth of effort. In the work described in Part II, I have been concerned largely with working toward a standard set of tools that makes compiler verification as easily as we realistically can expect it to be. The result is embodied in an open source software package called Lambda Tamer.

Chapter 6 presents my new methodology for formalizing high-level programming languages in Coq, a prerequisite for tool certification. Two main design decisions distinguish my approach from all prior formal verification work that I'm aware of. First, I use *dependently-typed abstract syntax* to encode typing rules in the abstract syntax trees of languages. This makes it possible to validate that compilation passes are *type-preserving* without writing any "proofs" of the usual kind. Second, I use a kind of *denotational semantics* to express what programs mean. Specifically, a language's dynamic semantics is a function compiling its terms into CIC, Coq's "native" programming language. The translation itself is a term of CIC, a critical ingredient that enables us to prove formal theorems about how code translations affect meaning.

Chapter 7 presents the total correctness certification of a type-preserving compiler from simply-typed lambda calculus to an assembly language. The compiler is *type-preserving* because static type information is threaded through the compiler from the source level to the resulting assembly programs, where it is used to build tables telling a garbage collector which registers to treat as roots. As a result, values stored in registers need no "dynamic typing" scheme along the lines of tagging or boxing. I present the particular intermediate languages that I use and the main ideas of their semantics. I spend some time discussing interesting aspects of the certification process, including a surprising opportunity to use a very simple heuristic decision procedure to prove many lemmas automatically.

Chapter 8 goes into detail on a particular tool introduced in Chapter 7, the AutoSyntax system. Today's standard typing judgment definition style suggests many "obvious," generic lemmas, such as how weakening and permutation of typing contexts don't break existing typing facts. Proving these lemmas again for each new type system can be very tedious. Following the formalization approach from Chapter 6, the situation is even worse, as typing judgments are combined with syntax; any change to the context in which a term is considered requires a change in the term's representation. I developed the AutoSyntax system to automate the generation of functions for reorganizing terms to use in new contexts, as well as to prove *automatically* lemmas that characterize how these reorganizations affect terms' denotational meanings. This code and proof generation is implemented *in CIC itself*, so that static typing guarantees valid code generation, using a technique called *proof by reflection*. The technique generalizes to other problems in generic programming.

One final note before diving into the case studies: This dissertation contains no appendices with painstakingly detailed proofs. All of my proofs are formalized in Coq, so I've felt free to focus here on providing pedagogically useful sketches, often including descriptions that aren't correct in full formality. The proofs can always be looked up online. I'll repeat the precise URLs again in the contexts of the individual projects, but here they are in one place. All of the code is under CVS version control at SourceForge, where it should be preserved indefinitely with full version history. Any references to the code in this dissertation should be taken to refer to the most recent CVS versions as of August 15, 2007.

Project	**URL**
Certified program verifiers	`http://proofos.sourceforge.net/`
Certified compilers	`http://ltamer.sourceforge.net/`

Part I

Certified Program Verifiers

Chapter 4

Programming with Specifications in Coq

Before discussing certified program verifiers in particular, I will use this chapter to introduce the basics of programming with specifications and proofs in Coq. In Section 4.1, I demonstrate how to combine "values" and "proofs" in unified packages, and I show how Coq's *extraction* facility neatly erases proof components so that they don't contribute to runtime inefficiency. Section 4.2 introduces Coq's module system and provides some general motivation and recipes for effective composition of certified software components.

4.1 Types and Extraction in Coq

The most basic Coq type for mixing programs and proofs is the *subset type* of the standard library:

$$\begin{array}{l}\mathsf{Inductive\ sig}\ (T : \mathsf{Set})\ (P : T \to \mathsf{Prop}) : \mathsf{Set} := \\ \quad \mid \mathsf{exist} : \forall x : T.\ P\ x \to \mathsf{sig}\ T\ P\end{array}$$

A type $\mathsf{sig}\ T\ (\lambda x : T.\ P)$ would more commonly be written $\{x : T \mid P\}$. It is often thought of as "the subset of the values of type T that satisfy predicate P." In type theory, it can be described more concretely as the type of pairs of values (of type T) and proofs that they satisfy a predicate (of type $T \to \mathsf{Prop}$).

Here we use an inductive definition of a family of Sets. I go beyond the simplified formalism from Chapter 2, where the only types of types were the Type_is. To support the extraction facility that I will introduce shortly, Coq features two other *sorts* besides Type: Set and Prop. The high-level intuition is that runnable programs with computational content belong to Set, while mathematical proofs belong to Prop. The types of programs are introduced with $\mathsf{Inductive}$ definitions with Set specified immediately before the :=, while propositions (i.e., the types of proofs) are introduced with Prop in that position.

Subset types are the basic ingredient in folding functions' preconditions and post-conditions into their types. However, we need another key ingredient to support certified programming language tools. We need specification types that admit the possibility of *failure*. I'll lead into the first example of this with a definition for the Coq version of the polymorphic `option` type familiar to ML programmers (and Haskell programmers as `Maybe`):

$$\begin{array}{l}\mathsf{Inductive\ option}\ (T : \mathsf{Set}) : \mathsf{Set} := \\ \quad \mid \mathsf{Some} : T \to \mathsf{option}\ T \\ \quad \mid \mathsf{None} : \mathsf{option}\ T\end{array}$$

Now consider this slight modification of option's definition:

$$\begin{array}{l}\mathsf{Inductive\ poption}\ (P : \mathsf{Prop}) : \mathsf{Set} := \\ \ \ \mid \mathsf{PSome} : P \rightarrow \mathsf{poption}\ P \\ \ \ \mid \mathsf{PNone} : \mathsf{poption}\ P\end{array}$$

Here I've changed the *argument type* of the polymorphic poption type to Prop, but left the *result type* the same at Set. A poption is a package that might contain a proof of a particular proposition or might contain nothing at all. The interesting thing about it is that, while it may contain a proof, it itself exists *as a program*. A helpful way to think about poption is as the rich return type of a potentially incomplete decision procedure that either determines the truth of a proposition or gives up. Such types will show up often in describing the building blocks of a system that tackles an undecidable problem like program verification.

As a concrete example, consider this function in concrete Coq syntax that determines if its argument is even:

```
Definition isEven : forall (n : nat), poption (even n).
  refine (fix isEven (n : nat) : poption (even n) :=
    match n return (poption (even n)) with
      | O => PSome _ _
      | S (S n) =>
        match isEven n with
          | PSome pf => PSome _ _
          | PNone => PNone _
        end
      | _ => PNone _
    end); auto.
Qed.
```

I'm using a lot of Coq notation here, but only a few details are relevant. First, the type of `isEven` is given as a dependent function type, where Coq uses `forall` in place of the more usual Π. A type `forall (x :  T1), T2` describes functions taking arguments of type `T1` and returning results of type `T2`, where the variable `x` is bound in `T2`, providing the opportunity for the function's result type to depend on the *value* of the argument.

Second, we provide a *partial* implementation for the function. We don't want to fill in the proofs manually; as Coq is designed for formalizing math, we rightfully expect that it can do this dirty work for us. By using a `Definition` command (terminated with a period) without providing an expansion for our new definition, we declare that we will construct this value with Coq's *interactive proof development mode.* In this mode, proof goals are iteratively refined into subgoals known to imply the original, until all subgoals can be eliminated in atomic proof steps. Individual refinements are expressed as *tactics*, small, untyped programs in the language that Coq provides for scripting proof strategies [29]. Theorem proving with tactics isn't my focus in this work, so I will just describe the two simple tactics that I've used in the example.

At any stage in interactive proof development, the goal is expressed as a search for a term having a particular type. The `refine` tactic specifies a *partial* term; it contains underscores indicating holes to be filled in, and we believe that there is some substitution for these holes that leads to a term of the proper type. Some holes are filled in automatically using standard type inference techniques, while the rest are added as *new subgoals in the proof search.*

In the use of `refine` in the example, I suggested a recursive function definition, filling in all of the computational content of the function and leaving out the details of constructing proofs. The holes standing for proofs turn out to be the only ones that Coq doesn't fill in through unification, and I invoke the `auto` automation tactic to solve these goals through Prolog-style logic programming.

We can make the code nicer-looking through some auxiliary definitions and by extending Coq's parser, which is built on "camlp4," the Caml Pre-Processor and Pretty Printer:

```
Definition isEven : forall (n : nat), [[even n]].
  refine (fix isEven (n : nat) : [[even n]] :=
    match n return [[even n]] with
      | O => Yes
      | S (S n) =>
        pf <- isEven n;
        Yes
      | _ => No
    end); auto.
Qed.
```

I introduce the syntax `[[P]]` for `poption P`, along with `Yes` and `No` for the `PSome _ _` and `PNone _` forms from the earlier example version. There's also the `pf <- isEven n; Yes` code snippet, which treats `poption` as a *failure monad* in the style familiar from Haskell programming [88]. The meaning of that code is that `isEven n` should be evaluated. If it returns `PNone`, then the overall expression also evaluates to `PNone`. If it returns `PSome`, then bind the associated proof to the variable `pf` in the body `Yes`. Here, it looks like the proof is not used in the body, but remember that `Yes` is syntactic sugar for a `PSome` with a hole for a proof. `refine` will ask us to construct this proof in an environment where `pf` is bound.

We construct terms like this to use in programs that we eventually hope to execute. With Coq, efficient compilation of programs is achieved through *extraction* to computationally equivalent OCaml code. With the right settings, our example extracts to:

```
let rec isEven (n : nat) : bool =
```

```
match n with
  | O -> true
  | S (S n) -> isEven n
  | _ -> false
```

Notice that the proof components have disappeared. In general, extraction *erases* all terms with sort Prop, leaving us with only the OCaml equivalents of Coq terms that we designated as "programs." Thanks to some subtle conditions on legal Coq terms, Coq can guarantee that the extraction of any Coq term in Set has the same computational semantics as the original.

Besides $\mathsf{poption}$, there is another type of similar flavor that will show up often in what follows. The $\mathsf{soption}$ type, which is an optional package of a value and a proof about that value, is defined as

$$\begin{array}{l}\mathsf{Inductive\ soption}\ (T : \mathsf{Set})\ (P : T \to \mathsf{Prop}) : \mathsf{Set} := \\ \ \ \mid \mathsf{SSome} : \forall x : T.\ P\ x \to \mathsf{soption}\ T\ P \\ \ \ \mid \mathsf{SNone} : \mathsf{soption}\ T\ P\end{array}$$

The type $\mathsf{soption}$ is the return type of a potentially incomplete procedure that searches for a value satisfying a particular predicate. For instance, a type inference procedure infer for some object language encoded in Coq might have the type

$$\forall e : \mathsf{exp}.\ \mathsf{soption\ type}\ (\lambda t : \mathsf{type}.\ \mathsf{hasType}\ e\ t)$$

We could then use this function in failure monad style with expressions like `t : pfT <- infer e; ...`, which attempts to find a type for `e`. If no type is found, the expression evaluates to `SNone`; otherwise, in the body `t` is bound to the value found, and `pfT` is bound to a proof that `t` has the property we need. The important difference of $\mathsf{soption}$ with respect to $\mathsf{poption}$ is that the value found by a function like infer is allowed to have computational content and is preserved by extraction, while the only computational content of a poption is a yes/no answer.

In the bulk of Part I, I'll use a more eye-friendly notation for these types. I'll denote poption P as $[\![P]\!]$ and soption T $(\lambda x : T.\ P)$ as $\{\!\{x : T \mid P\}\!\}$.

4.2 Modules and Correctness-by-Construction

The development strategy that I just outlined is certainly not the fastest route to certified functions. More traditionally, the programmer writes his function in a standard programming language with a relatively weak type system. He next uses a *verification condition generator* to produce a logical formula whose truth implies that the function satisfies its specification. Finally, he proves this verification condition, maybe even using an automated first-order theorem prover.

Such techniques tend to be geared towards views of software as static artifacts. While the tools driving traditional verifiers do often exploit modularity, support for this tends to be viewed as an optimization. With an appropriately efficient theorem proving black box, we would be just as happy to verify whole programs from scratch after each change to their source code. Systems like those in the ESC family [35] exemplify this sort of approach. They tend to be focused more on the "prove shallow properties of large programs" end of the verification spectrum, rather than the "prove deep properties of modestly-sized programs" tasks that concern me here.

There doesn't seem to be much hope of automating the tricky parts of a proof of verifier soundness. This means that a human must take an active role as proof architect.

Just as in software engineering, the inevitable consequence is that abstraction and modularity techniques are critical in enabling this human architect to do his job. It has been my experience that adding formal certification to a software project increases its difficulty by at least an order of magnitude, so many varieties of corner-cutting that we often use as programmers are no longer feasible. In particular, I've found it invaluable to have a strong specification associated with each piece of code throughout its entire existence.

In theory, classical verification supports this practice as well as dependent type systems do. Again, the problem appears in considering the evolution of certified programs. We want to write code by the repeated composition of smaller pieces. ESC-style verification can handle such programs in theory, but in practice, when it comes to deep properties, the automated prover behind the scenes will get stuck. What we really need is a convenient way of *composing proofs about software components in parallel to the components themselves.* I turn now to some examples demonstrating the value of this idea, which takes a variety of forms in idiomatic Coq code.

4.2.1 Dependent Types and Composition

First, a trivial example based on the isEven function from the last subsection. Imagine that we had elected to code isEven in a more traditional way, with no dependent types, leading to:

$$\mathsf{isEven} \quad : \quad \mathsf{nat} \rightarrow \mathsf{bool}$$

Now we want to build a new function to check that both components of a pair of natural numbers are even.

$$\mathsf{pairEven}(n_1, n_2) \quad = \quad \mathsf{isEven}(n_1) \ \&\& \ \mathsf{isEven}(n_2)$$

Next, we write down the obvious specification for pairEven:

$$\forall n_1, n_2 \in \mathbb{N}, \mathsf{pairEven}(n_1, n_2) = \mathsf{true} \rightarrow (\exists k_1 \in \mathbb{N}, n_1 = 2k_1) \wedge (\exists k_2 \in \mathbb{N}, n_2 = 2k_2)$$

How should we go about proving this theorem? We can prove a similar correctness theorem for isEven and use it to derive our goal straightforwardly. This verification style seems natural, but in practice it has a serious flaw. What happens as a certified program built in this way from components evolves? We have many *ad-hoc, implicit connections* between parts of programs and parts of proofs. For instance, the inductive argument used to prove the correctness of a recursive function will usually mirror the recursive structure of that same function. When we modify the function's source code, we need to hunt down the affected proof pieces and rewrite them, in a quite unprincipled way. In traditional software, we see such implicit dependencies as evidence of poor use of abstraction and modularity. There's no reason to be more forgiving when we move to certified programming.

To see a better approach, we return to the original version of isEven, having type:

$$\mathsf{isEven} \quad : \quad \forall n : \mathsf{nat}.\ [\![\exists k \in \mathbb{N}, n = 2k]\!]$$

Now we can write pairEven like this, using the monadic notation introduced in the last subsection:

$$\mathsf{pairEven}(n_1, n_2) \quad = \quad \begin{array}{l} pf_1 \leftarrow \mathsf{isEven}(n_1); \\ pf_2 \leftarrow \mathsf{isEven}(n_2); \\ \mathsf{Yes} \end{array}$$

The final “Yes” adds a logical subgoal to be proved using tactics.

Why is this version superior? First, we've made explicit the connection between each function and its correctness proof. This helps us in the same way that grouping related program pieces into modules or classes helps. It establishes a machine-checked convention that can help keep a lone developer honest and facilitate collaboration between multiple developers with a reduced need for informal documentation.

Second, the explicit program-proof connection makes it easier to express compiler/prover error messages with the traditional idiom of type errors. Proof-related error messages can signal not only a point within a proof with ad-hoc structure, but also the particular line of code in the program where that proof was built. When a program modification leads to incompatibilities in transitive dependencies, this kind of error message can be much more effective in determining what needs changing.

Another very compelling advantage isn't illustrated directly by our example. To build a new function by composing old functions, we had to do some new tactic-based proving. However, in many cases, dependent types let us write new certified code without ever writing anything we would call a “proof.” For instance, say we generalized the basic idea of $\mathsf{pairEven}$ with a library function:

$$\begin{array}{rcl}\mathsf{pairCheck} & : & \forall \tau : \mathsf{Set}.\ \forall P : \tau \to \mathsf{Prop}. \\ & & (\forall x : \tau.\ [\![P(x)]\!]) \to \forall y : \tau \times \tau.\ [\![P(\pi_1 y) \land P(\pi_2 y)]\!]\end{array}$$

$\mathsf{pairCheck}$ is a polymorphic function for lifting a test on values of type τ to a test on values of type $\tau \times \tau$, where it should be verified that the original property holds for both components of the pair. Now we can write a complete implementation of $\mathsf{pairEven}$ with:

$$\mathsf{pairEven} \quad = \quad \mathsf{pairCheck}\ \mathbb{N}\ (\lambda n.\ \exists k \in \mathbb{N}, n = 2k)\ \mathsf{isEven}$$

We get the same strong type that we came up with originally, without doing any more work than we would in traditional programming. Here I give the evenness predicate as an explicit argument that the programmer must type, but Coq can actually infer such arguments automatically, in this case from the type of isEven, giving us this simple definition:

$$\mathsf{pairEven} \quad = \quad \mathsf{pairCheck}\ \mathsf{isEven}$$

4.2.2 Modules and Proofs

The quest for "correctness theorems for free" has driven the design of much of the certified software architecture that I will present in this dissertation. In practice, most of the interesting cases do require some new proof inputs. It is beneficial to seek out reusable idioms for soliciting these proofs. Analogous problems in traditional programming have been solved quite elegantly by ML-style module systems [58]. Coq features a natural extension of such systems to its dependently-typed setting, and I make essential use of modules for structuring certified components.

As an example, we can revisit the standard example of functorized containers. A container family implemented with balanced trees requires a comparison operation over the type of keys that it stores. In ML, additional requirements on the comparator are stated in documentation but not checked. One reasonable requirement is that the comparator implement a total order. In Coq, we can make this requirement explicit and enable machine checking of it with this signature:

$$
\begin{array}{lll}
\mathsf{ORDERED} & = & \textbf{sig} \\
& & \begin{array}{rcl}
T & : & \mathsf{Set} \\
\mathit{leq} & : & T \rightarrow T \rightarrow \mathsf{bool} \\
\mathit{leq\_antisym} & : & \forall x, y : T.\ \mathit{leq}\ x\ y = \mathsf{true} \rightarrow \mathit{leq}\ y\ x = \mathsf{true} \\
& & \rightarrow x = y \\
\mathit{leq\_trans} & : & \forall x, y, z : T.\ \mathit{leq}\ x\ y = \mathsf{true} \rightarrow \mathit{leq}\ y\ z = \mathsf{true} \\
& & \rightarrow \mathit{leq}\ x\ z = \mathsf{true} \\
\mathit{leq\_total} & : & \forall x, y : T.\ \mathit{leq}\ x\ y = \mathsf{true} \vee \mathit{leq}\ y\ x = \mathsf{true}
\end{array} \\
& & \textbf{end}
\end{array}
$$

The signature resembles one you would see in an algebra textbook. Here we avoid using dependent types proper and rely instead on a more traditional axiom-based presentation. An equivalent formulation would leave out the three axioms and declare *leq* to have type $\{f : T \rightarrow T \rightarrow \mathsf{bool}\ \mid$ (* the axioms hold for every input pair *)$\}$. The choice of which style to use depends on the circumstances. When we care about multiple distinct properties of a single operation, the axiomatization approach is often more convenient. It's worth noting here that we can always transform an axiom-based implementation into one using only dependent types and no modules. In contrast to the situation for ML, the Coq module system adds no expressive power, but only convenience. Dependent record types subsume its features, though they can be significantly more clumsy to use.

We can define a signature of applicative sets, with some representative operations and one representative axiom:

$$\begin{array}{lcl}
\mathsf{SET} & = & \textbf{sig} \\
& & \begin{array}{rcl}
T & : & \mathsf{Set} \\
set & : & \mathsf{Set} \\
member & : & set \to T \to \mathsf{bool} \\
empty & : & set \\
add & : & set \to T \to set \\
add_ok & : & \forall S : set.\ \forall x : T.\ member\ (add\ S\ x)\ x = \mathsf{true}
\end{array} \\
& & \textbf{end}
\end{array}$$

We can write a Coq functor that transforms ORDEREDs into SETs:

Module MakeSet $(O$: ORDERED$)$: SET with Definition $T := O.T$
:= struct (* ... *) end

The body of the functor will define the types and operations just as they would be defined in ML. The axiom *add_ok* would probably be proved using tactics, drawing on the axioms from the parameter O as lemmas.

My main use of functors in this work has been for lowering a program verifier's level of abstraction. That is, such a functor takes as input a verifier and returns a new verifier at a lower level of abstraction, filling in the details that separate the two levels. The composition of these functors will give a method of transforming a high-level type-system description into a machine code analyzer. Naturally, we will require axioms establishing properties like type system soundness. Section 5.3 goes into detail on the component architecture that I use.

Chapter 5

A Certified Verifier Toolkit

5.1 Introduction

In this chapter, I will present a certified programming experiment with an interesting twist that adds an additional layer of reflection. The project deals with proving the correctness of programs that prove the correctness of programs. A proof of this kind provides correctness proofs "for free" for all the inputs the verified verifier can handle.

In particular, I have developed a framework for coding *certified program verifiers* for x86 machine code programs. The end results are executable programs that take x86 binaries as input and return either "Yes, this program satisfies its specification" or "I'm not sure." By virtue of the way that these verifiers are constructed using the Coq proof assistant, it is guaranteed that they are sound with respect to the unabstracted bit-level semantics of x86 programs. Yet this guarantee does not make development impractical; by re-using components outfitted with rich semantic interfaces, it's possible to whip together a certified verifier based on, for example, a new type system in a few hundred lines of code and an afternoon's time.

This work is related to two main broad research agendas: proof-carrying code and general software development techniques based on dependent types and interactive theorem proving. I devoted some space to motivating the latter subject in the introduction to this dissertation, but the former deserves some discussion here.

5.1.1 Applications to Proof-Carrying Code

The idea of certified program verifiers has important practical ramifications for foundational proof-carrying code (FPCC) [2]. Like traditional proof-carrying code (PCC) [67], FPCC is primarily a technique for allowing software consumers to obtain strong formal guarantees about programs before running them. The author of a piece of software, who knows best why it satisfies some specification that users care about, is responsible for distributing with the executable program a formal, machine-checkable proof of its safety. He might construct this proof manually, but more likely he codes in a high-level language that enforces the specification at the source level through static checks, allowing a *certifying compiler* [68] for that language to translate the proofs (explicit or implicit) that hold at the source level into proofs about the resulting binaries.

The original PCC systems were very specialized. A particular system would, for instance, only accept proofs based on a fixed type system. FPCC addresses the two main problems associated with this design.

First, traditional PCC involves trusting a set of relatively high-level axioms about the soundness of a type system. We would rather not have to place our faith in the soundness of so large a formal development, so FPCC reduces the set of axioms to deal only with the *concrete* semantics of the underlying machine model. If the soundness of a type system is critical to a proof, that soundness lemma must be proved from first principles.

The other problem is that a specialized PCC system is not very flexible. Typically, one of these systems can only check safety proofs for the outputs of a particular compiler for a particular source language. If you want to run programs produced with different compilers or that otherwise require fundamentally different proof strategies, then you will need to install one trusted proof checker or set of axioms for each source. This is far from desirable from a security standpoint, and FPCC fixes this problem by requiring all proofs to be in the same language and to use the same relatively small set of axioms. The axiomatization of machine semantics is precise enough that the more specific sets of axioms used in traditional PCC are usually derivable with enough work, if they were sound in the first place.

The germ of the project I'll describe comes from past work on improving the runtime efficiency of FPCC program checking [13]. Perhaps the largest obstacle to practical use of FPCC stems from the delicate trade-offs between generality on one hand and space and time efficiency of proofs and proof checkers on the other. Program verifiers like the Java bytecode verifier have managed to creep into widespread use almost unnoticed by laypeople, but naive FPCC proofs are much larger than the metadata included with Java class files and take much longer to check. It's unlikely that this increased burden would be acceptable to the average computer user.

Fundamentally, custom program verifiers with specialized algorithms and data structures have a leg up on very general proof-based verifiers. In our initial work on certified program verifiers, we proposed getting the best of both worlds by moving up a level of abstraction: allow developers to ship their software with specialized *proof-carrying verifiers*. These verifiers have the semantic functionality of traditional program verifiers and model-checkers, but they also come with machine-checkable proofs of soundness. Each such proof can be checked *once* when a certified verifier is installed. After the proof checks out, the verifier can be applied to any number of similar programs. These later verifications require no runtime generation or checking of uniform proof objects, which we found to be the major bottleneck in previous experience with FPCC. Our paper [13] presents performance results showing an order of magnitude improvement over all published verification time figures for FPCC systems for Typed Assembly Language [66] programs, by using a certified verifier. The verifier had a complete soundness proof, so no formal guarantees were sacrificed to win this performance.

The main problem that we encountered was in the engineering issues of proof construction. We used a more or less traditional approach to program verification in proving the soundness of our verifiers, writing them in a standard programming language and extracting verification conditions [32] that imply their soundness. Keeping the proof developments in sync with changes to verifier source code was quite a hassle. We also found that the structure of the verifier program and its proof were often very closely related, leading to what felt like duplicate work. I decided to try investigating what could be gained by writing verifiers from the start in a language expressive enough to encode verifier soundness in its type system.

5.1.2 Contributions

In the remainder of the chapter, I will describe my approach to the modular development of certified program verifiers. The key novelty is the use of dependent types in the "programming" part of development and in conjunction with Coq's ML-style module system, using the tools introduced in the last chapter. The end result is a set of components with rich interfaces that can be composed to produce a wide range of verifiers with low cost relative to the strength of the formal guarantees that result.

I'll begin by giving some preliminary background on the FPCC problem setting. Drawing on these definitions, I describe the design and implementation of a library to ease the development of certified verifiers via functors with rich interfaces. Next, I describe a particular completed application of that library, a memory safety verifier for machine code programs that use algebraic datatypes. I conclude by comparing with related work and summarizing the take-away lessons from the experience.

5.2 Preliminaries

5.2.1 Problem Formulation

The goal of this work is to support the verification of safety properties of executable x86 machine code programs. I've opted to simplify the problem by focusing on a single safety policy, where the safety policy simply forbids execution of a special "Error" instruction. As in model-checking, many interesting safety policies can then be encoded with assertion checks that execute "Error" on failure.

The first task is to define formally the semantics of machine code programs. The style is standard for FPCC [2], but I summarize the formalization here to make it clear exactly what a successful verification guarantees.

$$
\begin{array}{rrcll}
\textit{Machine words} & \mathsf{word} & w & ::= & 0 \mid 1 \mid \ldots \mid 2^{32}-1 \\
\textit{Registers} & \mathsf{reg} & r & ::= & \mathsf{EAX} \mid \mathsf{ESP} \mid \ldots \\
\textit{Flags} & \mathsf{flag} & f & ::= & \mathsf{Z} \mid \ldots \\
\\
\textit{Register files} & \mathsf{regFile} & R & = & \mathsf{reg} \rightarrow \mathsf{word} \\
\textit{Flag files} & \mathsf{flagFile} & F & = & \mathsf{flag} \rightarrow \mathsf{bool} \\
\textit{Memories} & \mathsf{memory} & M & = & \mathsf{word} \rightarrow \mathsf{byte} \\
\\
\textit{Machine states} & \mathsf{state} & S & = & \mathsf{word} \times \mathsf{regFile} \times \mathsf{flagFile} \times \mathsf{memory} \\
\\
\textit{Instructions} & \mathsf{instr} & I & ::= & \mathsf{ERROR} \mid \mathsf{MOV}\ r,\ [r] \mid \mathsf{JCC}\ f,\ w \mid \ldots \\
\\
\textit{Step relation} & & \mapsto & : & \mathsf{state} \rightharpoonup \mathsf{state}
\end{array}
$$

The main thing to notice is that the semantics follows precisely a conservative subset of the programmer-level idea of the "real" semantics of x86 machine code. I've chosen a subset of x86 instructions that is sufficient to allow many interesting programs and only included in the semantics those aspects of processor state needed to support those instructions.

The various elements of the formalization follow from the official specification of the x86 processor family [22], with the exception of the ERROR instruction added to model the safety policy. I'll briefly review the different syntactic classes and definitions before continuing.

A machine state consists of a word for the program counter, giving the address in memory of the next instruction to execute; a register file, giving the current word value of every general purpose register; a boolean valuation to each of the flags, which indicate conditions like equality and overflow relevant to the last arithmetic operation; and a memory, an array of exactly 2^{32} bytes indexed by words. The instructions are a subset of the real x86 instruction set, with the addition of the ERROR instruction.

A small-step transition relation $\mapsto$ describes the semantics of program execution. One transition involves reading the instruction from memory at the address given by the program counter and then executing it according to the x86 instruction set specification. Actually, $\mapsto$ is a partial function; it fails to make progress if the instruction loaded is ERROR. In this way, violations of the safety policy are encoded with the usual idiom of the transition relation "getting stuck." A "production quality" implementation would no doubt keep the real semantics separate from a library of safety policies, but the design decision I made simplifies my formalization, and the main interesting issues therein are the same between the two approaches.

We can define what it means for a machine state to be *safe* with this co-inductive inference rule:

$$\frac{S \mapsto S' \quad \mathsf{safe}(S')}{\mathsf{safe}(S)}$$

This is defined using Coq's facility for co-inductive judgments [37], which may have infinite derivations that are well-formed in a particular sense. Infinite derivations are important here for non-terminating programs.

The last ingredient is a means to connect a program to the first machine state encountered when it is run. Assume the existence of a type $\mathsf{program}$ and a function $\mathsf{load} : \mathsf{program} \rightarrow \mathsf{state}$. Concretely, $\mathsf{program}$ is a particular file format that GCC will output, and load expresses the algorithm for extracting the initial contents of memory from such a file, zeroing out registers and flags, and setting the program counter to the fixed address of the program start. To simplify reasoning while still remaining faithful to real semantics, I deal with programs that run "on a bare machine" with no operating system, virtual memory, etc.; and in fact the programs really do run as such in an emulator.

We've now established enough machinery to define formally the correctness condition of a certified verifier. A certified verifier is any value of the type:

$$\forall p : \mathsf{program}.\ [\![\mathsf{safe}(\mathsf{load}(p))]\!]$$

The type of the extracted function is $\mathsf{program} \rightarrow \mathsf{bool}$. By the soundness of extraction, we know that the value of the function on an input p is a boolean whose truth implies the safety of the program. Thus, if p is unsafe, the function must return false, and we can take a return of true as conclusive evidence that p is safe. A trivial certified verifier implementation is one that always returns false, but this is an issue of completeness, not soundness, to be dealt with through testing. Alternatively, one could craft a specialized declarative proof or type system, as is common in Typed Assembly Language, and prove that the verifier enforces exactly its rules, though I haven't done this to date. This isn't so urgent a requirement as it may sound, as a formally-defined decision procedure is already quite close to a declarative proof system.

5.2.2 An Example Input

The techniques I describe in this chapter are meant to be used with certifying compilers. However, since I chose to analyze machine code following a relatively simple type discipline not supported by any existing certifying compiler, it seemed more expedient to simulate the operation of a hypothetical compiler in producing test cases. To provide

```
#include "intlist.h"

BEGIN_FUNC(70);
STACK_TYPE(4, CUSTOM(LIST(INT)));
END_FUNC();
int int_list_len(list *ls) {
  int len = 0;

  BEGIN_CUTPOINT(50);
  REG_TYPE(ESP, STACK(16));
  REG_TYPE(EBP, STACK(12));
  STACK_TYPE(4, CUSTOM(LIST(INT)));
  STACK_TYPE(8, RETPTR);
  STACK_TYPE(12, OLDEBP);
  END_CUTPOINT();
  for (;;) {
    if (ls) {
      ++len;
      ls = ls->next;
    } else {
      BEGIN_CUTPOINT(50);
      REG_TYPE(ESP, STACK(16));
      REG_TYPE(EBP, STACK(12));
      STACK_TYPE(8, RETPTR);
      STACK_TYPE(12, OLDEBP);
      END_CUTPOINT();
      return len;
    }
  }
}
```

Figure 5.1: `length.c`

```
#define EAX "0"
#define ECX "1"
#define EDX "2"
#define EBX "3"
#define ESP "4"
#define EBP "5"
#define ESI "6"
#define EDI "7"

#define TOP "0"
#define CONST(n) "1, " #n
#define STACK(n) "2, " #n
#define OLDEBP "3"
#define RETPTR "4"
#define CUSTOM(n) "5, " n

#define BEGIN_FUNC(ss) __asm__("0:;.section .cutpoints; .int 1, 0b, " #ss)
#define END_FUNC() __asm__(".int 0; .section .text")

#define BEGIN_CUTPOINT(ss) __asm__("0:;.section .cutpoints; .int 2, 0b, " #ss)
#define END_CUTPOINT() __asm__(".int 0; .section .text")

#define REG_TYPE(r, t) __asm__(".int 1, " r ", " t)
#define STACK_TYPE(r, t) __asm__(".int 2, " #r ", " t)
```

Figure 5.2: `firstorder.h`

```
#include "firstorder.h"

#define INT "0"
#define LIST(t) "1, " t
#define NELIST(t) "2, " t

typedef struct list {
  void *data;
  struct list *next;
} list;
```

Figure 5.3: `intlist.h`

an idea of the real input format that these verifiers will be dealing with, I've included in Figure 5.1 a concrete listing of one of my test cases. This is meant to be used with one of the verifiers produced using the component architecture that I will describe in the next section: a verifier based on a simple type system of integers and lists.

This C source file, `length.c`, uses GCC-specific code to annotate the eventual machine code with metadata. This metadata can be thought of as consisting of a precondition for each basic block of the program. The header files shown in Figure 5.2 (generic metadata support) and Figure 5.3 (type system-specific support) define macros that emit inline assembly to save binary-encoded metadata in a special section of program binaries.

The arguments of the `BEGIN_FUNCTION` and `BEGIN_CUTPOINT` macros are stack size bounds, used to support usage of a stack discipline without dynamic overflow checks, which can be trickier to reason about. `REG_TYPE` annotations assign types to registers, and `STACK_TYPE` annotations assign types to stack slots numbered by offset from the stack pointer in effect at entry to the current function. The various uses of literal stack sizes and offsets are an example of mixing the C level of abstraction with the machine code level of abstraction. The example code listing should be thought of as a piece of machine code that we write using C as a macro assembler to automate the parts of low-level layout that it can easily be coerced to perform, without hesitating to calculate some offsets ourselves where necessary, based on knowledge of how GCC operates.

After sketching this hairy, ad-hoc scheme, it's worth pointing out that there are no formal proofs about it. Rather, an uncertified piece of OCaml code is responsible for reading this data and putting it into a nice, purely functional format at verification time.

The certified Coq code then picks up this description and uses it as a "hint" in verification. A similar scheme could be used for other sorts of verifiers, including more ambitious approaches that require the inclusion of arbitrary logical formulas or even proof snippets with program binaries, although the particular component architecture that I will present next isn't very well-suited to many such styles of verification.

5.3 Components for Writing Certified Verifiers

The final goal of the case study I'm presenting here was to produce a certified x86 machine code memory safety verifier that supports general product, sum, and recursive types, which I'll call MemoryTypes. It would have been possible to write this verifier monolithically, but I thought it would be more interesting and useful to do it in stages, writing re-usable components with rich interfaces to handle different parts of verification and allow later components to reason at increasingly high levels of abstraction.

A word about generality The rest of this chapter can be thought of as presenting two distinct contributions. One is the particular architecture that is the subject of this section. I present a particular set of components that can be put to good use crafting verifiers for a particular class of programs.

That class is far from universal, however. The architecture I'll describe can't be used in the verification of programs with first-class code pointers, to name just one major weakness. Rather, this architecture is intended as a case study in support of the second contribution, which is a methodology for constructing certified program analysis tools.

I believe that the techniques I present, based on dependent types and abstraction-

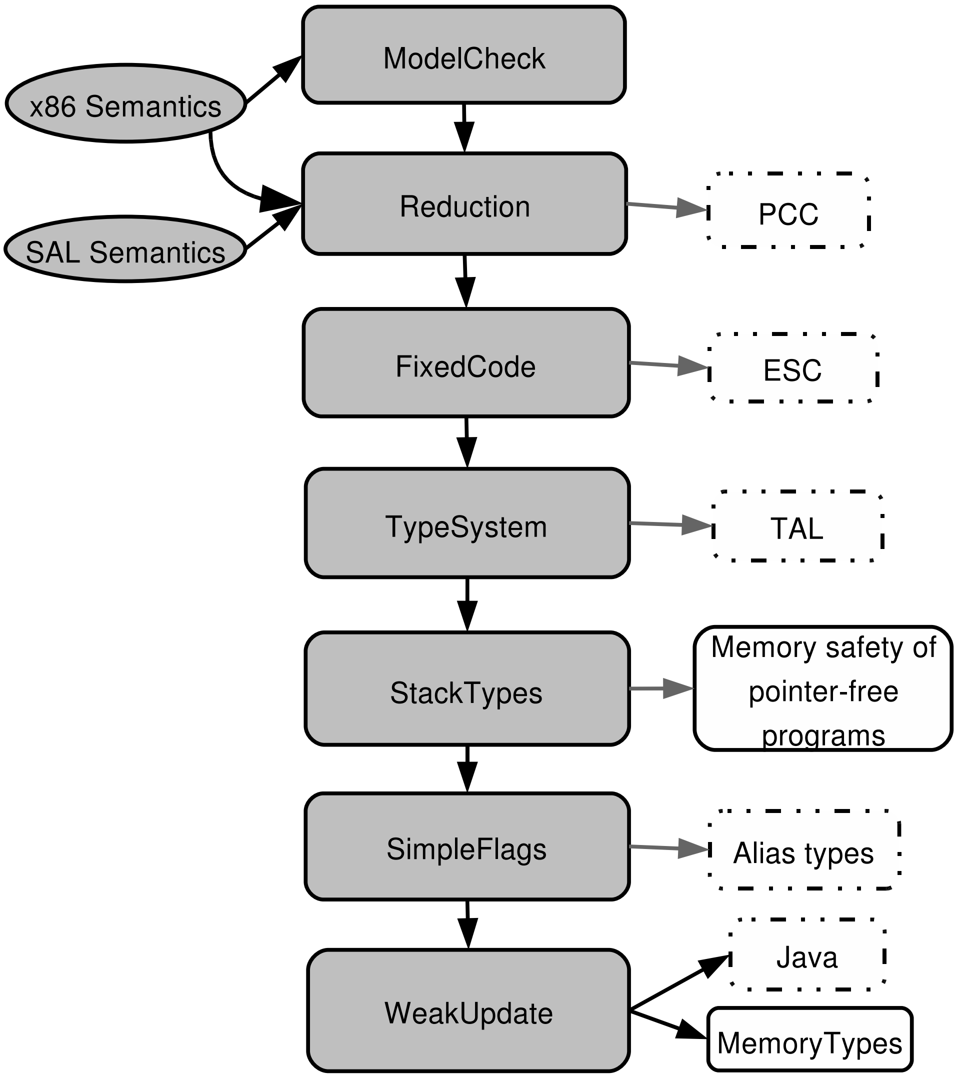

Figure 5.4: A component structure for certified verifiers

spanning functors, could be put to good use in designing a system to make it easier to implement, say, traditional typed assembly languages [66]. I have no empirical evidence to present here for that claim, but, in any case, the results here are interesting in their own right. The world of certified verifiers is a new one, and, as is usually the case in domains like this that depend on formal proofs, it's much easier to draw conclusions about what works and what doesn't by starting with relatively simple examples. So, in that light, I invite the reader to consider the final verifier produced in this work as a case study within a case study.

Introduction to the architecture The component structure that I present here is born of necessity; a layered decomposition of verifier structure or something like it is critical to making the overall task feasible. As traditional software built from many simple pieces can become unmanageably complex, the problem is only exacerbated when formal correctness proofs are required, since now even the "simple" pieces can involve non-trivial proofs. The component structure I've settled on has been designed not just to support effective programming, but also effective proof construction, by minimizing the need for repeated work. The issues and complexities specific to my domain of machine code verifiers are probably not clear to readers who don't have experience in that field, but I hope that the following walk-through of the steps in my solution can shed some light on them. The important question at each stage of this abstraction hierarchy is "How hard would it be to develop and maintain a new verifier (with a soundness proof) handling all of the hidden lower-level details?". It's also true that I'll be drilling down to a significant level of detail in this section. I invite the overwhelmed reader to look ahead to the light at the end of the tunnel in

Section 5.4, where I show how all of this machinery pays off in making it very easy to build a new certified verifier.

Figure 5.4 presents the particular component structure that I settled on. An arrow from one component to another indicates that the target component of the arrow builds on the source component. Ovals represent logical theories that are used in the correctness conditions of other modules, such as the x86 semantics. Boxes stand for components that contribute code to the extracted version of a verifier; i.e., they contain implementations of verifiers at particular levels of abstraction, along with the associated correctness proofs. Solid boxes (like the WeakUpdate component) are best viewed as library components, while transparent boxes represent certified verifiers, the final products. I include a number of verifier boxes with dashed borders. These stand for hypothetical verifiers that I haven't implemented but that I believe would best be constructed starting from the components that connect to them in the diagram.

The basic paradigm here is that we use *functors* to transform *abstractions* to "lower levels;" that is, we enrich abstractions with explicit handling of details that they had previously considered to be tracked by "their environments." At the end of this process, an abstraction that makes the simplifying assumptions typical of high-level type systems will have "learned" to track all of the pertinent minutiae of x86 machine code. For our purposes here, an "abstraction" can be thought of as a perspective on how to "think about" the execution of programs, assuming that certain kinds of (abstraction-specific) information are available from an oracle. The concept of abstraction will remain informal, as different verification strategies suggest different categories of abstraction. The unifying similarity across all cases will be that each abstraction provides a language for describing program states and an account of how these states are affected by different concrete operations.

I will describe each library module in detail in the following subsections, but I'll start by providing an overview of the big picture.

- The only module that belongs to the trusted code base is the **x86 Semantics**, the basic idea of which I presented in Section 5.2.1.

- **ModelCheck** provides the fundamental method of proving theorems about infinite state systems through exhaustive exploration of an appropriate abstract state space; or, since x86 states are finite in reality and in my formalization, proving theorems about intractably large state spaces through exhaustive exploration of smaller abstract state spaces.

- The CISC x86 instruction set involves lots of complications that one would rather avoid as much as possible, so I do most verification on a tiny RISC instruction set to which I reduce x86 programs. **SAL semantics** defines the behavior of this Simplified Assembly Language.

- **Reduction** enables multiple steps of abstraction: model checking an abstraction of an abstraction of a system suffices to verify that system. In the chain of component uses for MemoryTypes, Reduction is used to do model checking on the SAL version of an x86 program. One way of viewing traditional PCC approaches is that they apply proof-checking on the result of a reduction to whatever internal format they use to represent programs.

- **FixedCode** deals with a basic simplification used by most program verifiers, which is that a fixed region of memory is designated as code memory, and that memory region cannot be modified in any run of the program. General FPCC frameworks in theory support verification of self-modifying programs, but we usually want to work at a higher level of abstraction. FixedCode's level of abstraction would be appropriate for an adaptation to machine code level of traditional verification in the style of Extended Static Checking [35].

- **TypeSystem** provides support for model checking where the primary component of an abstract state is an assignment of a type to every general purpose machine register. This would be a good starting point for traditional Typed Assembly Language [66, 65], which handles stack and calling conventions with its own kind of stack types...

- ...but for most verifiers, **StackTypes** would be the module to use next. It takes as input a type system ignorant of stack and calling conventions and produces a type system that understands them. An application of StackTypes to a trivial type system gives us a verifier capable of checking memory safety of simple C programs that don't use pointers.

- **FlagTypes** handles tracking of condition flag values relevant to conditional jumps. This is critical for verifying programs that use pointers that might be null, general sum types, or any of a large variety of type system features. FlagTypes would be a reasonable starting point for a verifier based on alias types [82] or some other way of supporting manual memory management...

- ...but with automatic memory management, **WeakUpdate** provides a much more convenient starting point. WeakUpdate is used with type systems that have a notion of a partial map from memory addresses to types, where this map can only be extended, never modified, during a program execution. Though addresses can usually change types when storage is reclaimed, this is handled by, e.g., a garbage collector that is verified using different methods. WeakUpdate would also provide a good foundation for machine code-level verification of programs compiled from Java source code.

In general, each of these arrows between rounded boxes in Figure 5.4 indicates a functor translating a verifier at the target's level of abstraction to a verifier at the source's level. These functors are used as in the example in Section 4.2. For instance, for the arrow between TypeSystem and StackTypes, we have the form of the earlier example with `ORDERED` changed to `STACK_TYPE_SYSTEM`, the output signature of the functor changed from `SET` to `TYPE_SYSTEM`, and the functor's innards assembling a richer type system by extending that presented by its input module.

I will now describe the most important aspects of the interfaces and implementations of each of these reusable library components. I avoid describing how the actual proofs are constructed, focusing instead on component interfaces and a bird's-eye view of an overall structure that I've found to work in practice. Nonetheless, there are many important engineering issues in proof construction, and doing the subject justice would require another article of its own. The main thing to keep in mind through the following sections is that every piece *is* supported by Coq proofs of the relevant properties, and that I was able to construct these proofs using the techniques sketched in Chapter 4.

Module Type MC_ABSTRACTION.

Module	Mac	:	MACHINE.
Parameter	$absState$	:	Set.
Parameter	$context$	:	Set.
Parameter	$\vdash_S$	:	$context \to Mac.state \to absState \to$ Prop.
Definition	mcstate	:=	$absState \times$ list $absState$.
Parameter	$init$	:	$\{states$: list $(absState \times$ list $absState)$ $\mid \exists\Gamma : context, \exists\alpha : absState,$ $(\alpha, \mathsf{nil}) \in states \wedge \Gamma \vdash_S Mac.start : \alpha\}$.
			(* Auxiliary definitions *)
Parameter	$step$	:	$\forall hyps$: list $absState$. $\forall\alpha : absState$. $\{\!\{succs$: list $absState \mid$ progress(α) $\wedge$ preservation$(\alpha, hyps, succs)\}\!\}$.

End MC_ABSTRACTION.

Module Type VERIFIER.

Module	Mac	:	MACHINE.
Parameter	$check$	:	$\forall p : Mac$.program. $[\![Mac$.safe$(Mac$.load$(p))]\!]$.

End VERIFIER.

Module ModelCheck $(A$: MC_ABSTRACTION$)$
: VERIFIER with Module $Mac := A.Mac$.
(* Definitions *)
End ModelCheck.

Figure 5.5: Input and output signatures of ModelCheck

5.3.1 ModelCheck

Figure 5.5 shows the input and output signatures of the first component in the pipeline, ModelCheck. Since such figures can be hard to digest, in the rest of this section I've opted to follow a different convention. I won't discuss the output signatures of functors, as they just match up with the inputs of earlier pipeline stages, in general. I'll present input signatures member-by-member, interleaved with explanatory text in the main body of this section. Moving through the subsections describing the different functors, the input signatures will build on their predecessors. Thus, I will only describe *new* members; old members that don't draw explicit mention should be assumed to stay the same as in the previous cases. When an old member is mentioned again later, the new definition should be taken to replace the original.

The definitions of the fixed component input signatures will alternate with illustrative examples. To help distinguish which symbols are which, names of signature pieces ("formal parameters") will be written in *italics*, while names of pieces of examples ("actual parameters") will be written in a serif font. Following the same "interface vs. implementation" convention, standard abbreviations that are included in signatures will also have serif names.

The astute reader will be able to notice cases where, following this convention literally, some of the signatures that would be assigned to the different components will be incomplete or inconsistent. I've chosen what to include with pedagogic effectiveness in mind. A pleasant consequence of describing a completely formal piece of theory is that I need not worry about this sort of omission as much as usual, since the formalization is available in full on the Web. (See Section 5.5.) I hope that the result in these pages manages to express the key ideas of the components while not taxing the reader's patience.

An input to ModelCheck specifies a particular machine semantics *Mac*, implemented as a module ascribing to a particular common `MACHINE` signature. An abstraction for this machine is defined with an implementation of a signature that I will describe piece by piece.

Before going into these details, I note that this formalization of model-checking is specific to "first-order" uses of code pointers. Each point in the abstract state space can have any number of known code pointers that it is allowed to jump to, but the descriptions of these code pointers can't themselves refer to other code pointers. The formulation I give is expressive enough to handle, for instance, standard function call and exception handling conventions. Naturally, this design decision precludes the easy handling of functional languages, but one would simply write another component to serve as a starting point there; and there are plenty of interesting issues in this restricted setting, related to data structures and other program features.

Now I will give the high level picture of what an abstraction is and what properties it must satisfy. The fundamental piece of an abstraction is its set *absState* of abstract states. These will be the constituents of the state spaces explored at verification time.

$$absState \; : \; \mathsf{Set}$$

Next, we have an abstraction relation:

$$\begin{aligned} context \; &: \; \mathsf{Set} \\ \vdash_S \; &: \; context \to Mac.state \to absState \to \mathsf{Prop} \end{aligned}$$

As per usual in abstraction-based model checking, we need to provide a relation characterizing compatibility of concrete and abstract states. $\vdash_S$ is a ternary relation serving in this role. It relies on an extra, perhaps unexpected, component, a set *context*. The basic idea behind the separation of abstract states and contexts is that abstract states will be manipulated in the extracted OCaml version of a verifier, while contexts will be used only in the proof of correctness and erased during extraction. Though we define contexts to have computational content, it will turn out that, if we use a naive extraction that preserves all definitions in sort Set, a dependency analysis starting from the resulting verifier's entry point will show that contexts aren't used. Coq's `Recursive Extraction` command achieves the same effect by outputting definitions lazily, as needed to export one root definition, saving us from the ridiculous situation of translating the entire Coq standard library on every extraction.

A canonical example of a context is a valuation to free type variables used in an abstract state:

$$\begin{array}{rrcl} \text{Type variables } V & \beta & & \\ \text{Types} & \mathsf{type} & ::= & \beta \mid \ldots \\ & \mathit{absState} & = & \mathsf{reg} \rightarrow \mathsf{type} \\ & \mathit{context} & = & V \rightarrow \mathsf{type} \end{array}$$

Contexts provide a sort of polymorphism that lets us check infinitely many different abstract states by checking a finite set of representatives. For instance, we check a finite set of abstract states containing type variables in place of checking the infinite set of all of their substitution instances. I use the mixfix notation $\Gamma \vdash_S s : \alpha$ to denote that, in context Γ, concrete state s and abstract state α are compatible; i.e., s belongs to α's concretization.

Now we need a way of computing an abstract state space that conservatively approximates the concrete state space. To describe ModelCheck and the functors that follow it, I will use a simple running example, based on elements of verifying simple compiled C programs. Some features that I present monolithically here will actually be added modularly by earlier functors, and I omit some details here necessary for real verification.

$$\begin{array}{lrcl}
\mathsf{type} & \tau & ::= & \mathsf{int} \mid \tau\ \mathsf{ptr} \mid \mathsf{retptr} \\
& \mathit{absState} & = & (\mathsf{word} \cup \{\mathsf{retptr}\}) \times (\mathsf{reg} \to \mathsf{type}) \\
& \mathit{context} & = & \mathsf{word}
\end{array}$$

As in the real final verifier of my case study, this is a type-based checker. Our types include untagged integers, pointers, and a distinguished type for a saved function return pointer.

Abstract states have two components. First, we have a representation of the program counter. This will either be a word, giving the known program counter value; or a distinguished retptr value, for a state immediately after jumping to the saved return pointer. The second abstract state component is the standard map from registers to types.

Our contexts are very simple here. There is only one piece of information needed in reasoning about the verifier's correctness but not in its algorithmic operation: the concrete value of the return pointer. We check functions with the return pointer treated symbolically, but our soundness proof refers to the context to show that the verification applies to any concrete function call that might be made.

In general, each element of a ModelCheck state space consists of one *absState* describing the current state and zero or more *hypotheses* (represented with a $\mathsf{list}\ \mathit{absState}$) describing other abstract states known to be safe.

$$\mathsf{mcstate} \quad = \quad \mathit{absState} \times \mathsf{list}\ \mathit{absState}$$

The canonical example of a hypothesis is a function call's return pointer. If verification inside a function ever reaches an abstract state compatible with the return pointer's hypothesis, then there is no need to explore that branch of the state space further.

Here is an example element of a ModelCheck state space. I overload set notations to work for lists, where a list is interpreted as the set of its elements. I also write $[r_1 \mapsto \tau_1, \ldots, r_n \mapsto \tau_n]$ to denote the function mapping each register r_i to type τ_i and all other registers to int.

$$((42, [\mathsf{EAX} \mapsto \mathsf{int\ ptr\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\}) : \mathsf{mcstate}$$

We have a standard state for the inside of a function body. The program counter is known to be 42, register EAX points to a pointer to an integer, and register EBX holds the saved return pointer. Our single hypothesis reflects the conditions under which it is safe to return from the function: The program counter must be restored to the saved value, and EAX must point to an integer.

Our next ingredients are the algorithmic pieces of a verifier. These are *init*, which calculates the roots of the state space; and *step*, which describes how to expand the visited state space by following the edges out of a single node. Starting with *init*:

$$\begin{array}{rcl} \mathit{init} & : & \{\mathit{states} : \mathsf{list}\ (\mathit{absState} \times \mathsf{list}\ \mathit{absState}) \\ & & \quad \mid \exists \Gamma : \mathit{context}, \exists \alpha : \mathit{absState}, \\ & & \quad\quad (\alpha, \mathsf{nil}) \in \mathit{states} \land \Gamma \vdash_S \mathit{Mac.start} : \alpha\} \end{array}$$

init provides a set (actually a list) of state descriptions, along with a guarantee that some abstract state compatible with the concrete initial state is included. The condition for *init* requires that some abstract state with no hypotheses (i.e., that makes no special

$$
\begin{array}{rrl}
\underline{0}: & \mathsf{MOV\ EAX},\ p & ((0, []), \{\}) \\
4: & \mathsf{CALL}\ 12 & ((4, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}]), \{\}) \\
\underline{8}: & \mathsf{JMP}\ 8 & ((8, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}]), \emptyset) \\
\\
\underline{12}: & \mathsf{MOV\ [EAX],\ EAX} & ((12, [\mathsf{EAX} \mapsto \mathsf{int\ ptr\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\}) \\
16: & \mathsf{JMP\ EBX} & ((16, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\}) \\
\\
\underline{20}: & \mathsf{MOV\ [EAX],\ EAX} & ((20, [\mathsf{EAX} \mapsto \mathsf{int\ ptr\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\}) \\
24: & \mathsf{JMP}\ 28 & ((24, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\}) \\
 & \ldots & \\
\underline{28}: & \ldots & ((28, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\}) \\
 & \ldots & \\
45: & \mathsf{JMP}\ 28 & ((45, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})
\end{array}
$$

Figure 5.6: Example program for Section 5.3.1

assumptions) is found among the initial states.

Consider the program (annotated with abstract states) in Figure 5.6 as an example. Let's say that the concrete initial state has the program counter, registers, and memory cells all initialized to 0. The program we want to verify has, in addition to start-up code at PC 0, two functions. The start-up code calls the first function and then enters an infinite loop.

Each function takes an int ptr ptr as an argument and returns an int ptr. Their entry points are PC's 12 and 20. The first function has just a single basic block, and the second function has an additional basic block starting at PC 28 and containing a loop. For the purposes of this example, we'll use a calling convention where the first argument to a function is passed in register EAX, and the return value is stored in the same register. We also use register EBX to store the saved return pointer. This determines the states on entry to the functions, and let's consider that the loop entry point (PC 28) assumes that EAX has been modified to contain an int ptr.

Here is a good set of initial states. There is one entry for each underlined program counter in Figure 5.6.

$$
\begin{array}{l}
\{((0,[]),\{\}),\\
((8,[\mathsf{EAX} \mapsto \mathsf{int\ ptr}]),\emptyset),\\
((12,[\mathsf{EAX} \mapsto \mathsf{int\ ptr\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]),\{(\mathsf{retptr},[\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\}),\\
((20,[\mathsf{EAX} \mapsto \mathsf{int\ ptr\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]),\{(\mathsf{retptr},[\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\}),\\
((28,[\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]),\{(\mathsf{retptr},[\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})\}
\end{array}
$$

As the discussion to follow shortly will show, *init* can't simply return a complete description of the initial concrete state, but must rather contain enough states that every point in the program's execution reaches one in finitely many steps. In this sense, *init* is like a pre-computed fixed point of an abstract interpretation. I could have required that *init* contain enough elements to describe *every* reachable concrete state, but that would just contribute to verification-time inefficiency, and we really only need to fix enough abstract states to cut every cycle in the abstract state space. In the implementation, *init* will be computed mostly by ML code. This code gets the information by reading annotations out of the binary being analyzed. It could just as easily use abstract interpretation to infer the information from a less complete set of annotations. Notice that an error in constructing the fixed point can only effect completeness, not soundness, so it's OK to implement this part outside of Coq.

We now turn to the *step* function. It has a complicated type with several pieces, so I'll present it in stages, using a series of predicates to be defined one at a time.

$$
\begin{array}{lcl}
step & : & \forall hyps : \mathsf{list}\ absState.\ \forall \alpha : absState.\\
 & & \quad \{\!\{ succs : \mathsf{list}\ absState \mid \mathsf{progress}(\alpha) \wedge \mathsf{preservation}(\alpha, hyps, succs) \}\!\}
\end{array}
$$

The overall form of *step* is simple enough. It is a dependent function taking as inputs the current hypothesis set and the current abstract state. Note that while *init*'s type is a standard subset type, *step* has an `soption` type that allows it the option of failing for any input. Naturally, this is important, since otherwise the implementation of ModelCheck would somehow need to produce a model checker that is able to prove any program safe!

When *step* succeeds, it returns the set of new states that should be explored before declaring the program safe. The subset type imposes some standard requirements on such a set of states. We have a *progress* condition, expressing that the program is able to make at least one execution step; and a *preservation* condition, expressing that every possible concrete state transition matches up with an abstract transition to a member of *succs*.

The progress condition is the simpler of the two:

$$\begin{aligned}\mathsf{progress}(\alpha) \quad = \quad & \forall s : \mathit{Mac.state}.\ \forall \Gamma : \mathit{context}. \\ & \quad \Gamma \vdash_S s : \alpha \rightarrow \exists s' : \mathit{Mac.state}, s \mapsto_{\mathit{Mac}} s'\end{aligned}$$

For every concrete state and context compatible with the current abstract state, execution must make at least one more step of execution.

The preservation condition is broken into two cases, roughly corresponding to regular and control-flow instructions.

$$\begin{aligned}\mathsf{preservation}(\alpha, \mathit{hyps}, \mathit{succs}) \quad = \quad & \forall s, s' : \mathit{Mac.state}.\ \forall \Gamma : \mathit{context}. \\ & \quad s \mapsto_{\mathit{Mac}} s' \wedge \Gamma \vdash_S s : \alpha \\ & \quad\ \rightarrow\ \exists \Gamma' : \mathit{context},\ \mathsf{regular}(\mathit{hyps}, \mathit{succs}, s', \Gamma') \\ & \quad\quad \vee\ \mathsf{controlFlow}(\mathit{hyps}, \mathit{succs}, s', \Gamma, \Gamma')\end{aligned}$$

We consider any concrete state transition with a source state compatible with α

in some context Γ. There must exist a new context Γ' such that one of the two subconditions $\mathsf{regular}$ and $\mathsf{controlFlow}$ applies. The first kind of preservation is the simple one, corresponding to most instructions in a program:

$$\begin{array}{rcl}\mathsf{regular}(hyps, succs, s', \Gamma') & = & \exists \alpha' : absState, \alpha' \in (hyps \cup succs) \\ & & \wedge\, \Gamma' \vdash_S s' : \alpha'\end{array}$$

When $\mathsf{regular}$ holds, we have a new abstract state α' that is either a hypothesis or one of the new states queued for exploration. In either case, the fact that α' describes s' in Γ' shows us that the work we've already done or queued to do will cover this concrete transition.

The definitions so far are sufficient to demonstrate the handling of two representative cases for our running example. Consider first the entry point:

$$((12, [\mathsf{EAX} \mapsto \mathsf{int\ ptr\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})$$

for the example's first function. The first instruction is $\mathsf{MOV\ [EAX],\ EAX}$, which dereferences the pointer stored in register EAX and writes the result back into EAX. Passed the given hypotheses and state, *step* could return:

$$\{(16, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}])\}$$

We establish the progress condition by looking up address 12 in the program code and verifying that it contains the encoding of a non-ERROR instruction. We establish preservation using the $\mathsf{regular}$ case, setting α' to be the sole member of our successor state set and keeping the original context Γ as Γ'. Proving $\Gamma' \vdash_S s' : \alpha'$ involves proving a typing condition for each register. We handle it trivially for the registers that haven't changed, and the case for

EAX is proven directly from the semantics of the MOV instruction and ptr types.

A more interesting case is a return from the function. Observe that the last instruction that we considered put the program into a state where it's ready for a return, and the next instruction JMP EBX does exactly that. The abstract state beforehand is:

$$((16, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})$$

The function *step* can return the empty set, since we will assume that all possible return states are queued for visitation at the times their calls are made. We establish progress in the same way as before. To show preservation, we exhibit this state as α':

$$(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])$$

The only changes from the starting state are that the program counter has changed to retptr and we have forgotten what we knew about the type of EBX. We exercise the first possibility in the condition $\alpha' \in (hyps \cup succs)$, as α' is exactly our return pointer hypothesis. Assuming reasonable definitions for the underlying typing judgment, it is easy to establish $\Gamma' \vdash_S s' : \alpha'$ when we set $\Gamma' = \Gamma$. A primary element of this reasonableness is that retptr should be interpreted as the singleton type of words equal to the return pointer saved in the context Γ.

The second kind of preservation is associated with direct jumps and function calls:

$$\begin{array}{rcl} \mathsf{controlFlow}(hyps, succs, s', \Gamma, \Gamma') & = & \exists hyps' : \mathsf{list}\ absState, \exists \alpha' : absState, \\ & & \quad (\alpha', hyps') \in \pi_1 init \\ & & \quad \wedge\ \exists \Gamma' : context, \Gamma' \vdash_S s' : \alpha' \\ & & \quad\ \ \wedge\ (\Gamma', hyps') \leq_S^{succs} (\Gamma, hyps) \end{array}$$

It requires that we look into the roots of the state space and find one $(\alpha', hyps')$ that is compatible with s'. The operator π_1 expresses projecting out the computational part of a subset type, in this case giving us the list of roots without the proof concerning them. Again we may provide a new context Γ'. However, this time not only do we need to guarantee $\Gamma' \vdash_S s' : \alpha'$, but we also need to be sure that every hypothesis in $hyps'$ describes a set of states that are all safe. That's the purpose of the final condition, which says that every hypothesis hyp' in $hyps'$ has a counterpart hyp among the current hypotheses and successor states, such that any concrete state described by hyp' in Γ' is also described by hyp in Γ. The notation $\leq_S$ expresses this requirement:

$$\begin{aligned}(\Gamma', hyps') \leq_S^{succs} (\Gamma, hyps) \quad = \quad & \forall h' \in hyps'.\ \exists h \in (hyps \cup succs), \\ & \forall s.\ \Gamma' \vdash_S s : h' \to \Gamma \vdash_S s : h\end{aligned}$$

This implication at first seems to be reversed from the natural order, but it makes sense in the light of standard function subtyping rules when we think of hypotheses as distinguished function arguments.

Another two example cases help illustrate the use of $\mathsf{controlFlow}$. We look first at a direct jump within a function. Let's start at PC 24, a point in the second function just after dereferencing the function's argument into EAX. We are in this abstract state:

$$((24, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})$$

We encounter a direct jump to program counter 28. Now we use the following singleton set of successor states and draw α' from it:

$$\{(28, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}])\}$$

We can verify that the pairing of this state with the current return hypothesis is indeed found among the elements of *init*. The compatibility conditions from $\mathsf{controlFlow}$ are verified trivially, since the starting abstract state differs from the result only in a change of program counter.

Finally, let's look at the entry point to the example program, which initializes EAX to a valid pointer of proper type, calls the function, and then loops forever at its return site. Before the call, the abstract state would look something like:

$$((4, [\mathsf{EAX} \mapsto \mathsf{int\ ptr\ ptr}]), \emptyset)$$

The $\emptyset$ shows that we have no hypotheses yet. Upon using a fictitious function call instruction that sets the program counter and EBX simultaneously, we would want to present a list of successor states like this one:

$$\{((8, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}]), \emptyset)\}$$

The single successor is for the return point. It is our responsibility as the caller to queue it for visitation, so that the function may assume it safe to return to. To prove preservation, we present an $(\alpha', hyps')$ that has gone through a point-of-view shift to the function's perspective:

$$((((12, [\mathsf{EAX} \mapsto \mathsf{int\ ptr\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})$$

It is easy to show $\Gamma' \vdash_S s' : \alpha'$ with Γ' set to 8, the address of the return site. The hypothesis safety condition also comes easily, establishing the safety of the single hypothesis by pointing out that an equivalent to it has been queued for visitation.

With these components provided to it, ModelCheck produces a standard model checker that uses the requested abstraction. This model checker performs a depth-first search through the state space, where the search terminates in every branch of the tree where preservation is shown through the controlFlow case, corresponding to a jump or call to one of the root states. The computational content of *init* and *step* determines the shape of the state space to explore.

This description reveals that ModelCheck doesn't *quite* produce a standard model checker. Rather, it produces a standard *validator* for the results of a traditional fixed point calculation. The calculation must have been performed ahead of time by a certifying compiler or some other non-certified (and untrusted) algorithm that relays its answer to the certified piece of the code through *init*, as described in Section 5.2.2.

5.3.2 Reduction

The literal x86 machine language is not ideal for verification purposes. Single instructions represent what are conceptually several basic operations, and the same basic operations show up in the workings of many instructions. As a result, a verifier that must handle every instruction will find itself doing duplicate work. In my implementation, I handle this problem once and for all by way of a component to model check a program in one instruction set by reducing it to a simpler instruction set.

The particular simplified language that I use is a RISC-style instruction set called SAL (Simplified Assembly Language), after a family of such languages used in traditional proof-carrying code work [67]. The main simplifications are the use of arbitrary arithmetic expressions, instead of separate instructions for loading a constant into a register, performing an arithmetic operation, etc.; and a new invariant that each instruction has a single effect on machine state. This latter property is accomplished by breaking instructions into multiple pieces responsible for the different effects.

Here's a brief summary of the language grammar:

$$
\begin{array}{rrcl}
\textit{SAL registers} & r_s & ::= & r \mid \mathsf{TMP}_i \\
\textit{Binary operators} & \circ & & \\
\textit{SAL expressions} & e & ::= & w \mid r_s \mid e \circ e \\
\textit{SAL instructions} & I_s & ::= & \mathsf{ERROR} \mid \mathsf{SET}\ r_s,\ e \\
& & & \mid\ \mathsf{LOAD}\ r_s,\ [e] \mid \mathsf{STORE}\ [e],\ e \mid \ldots
\end{array}
$$

I add a finite set of extra temporary registers, needed when a single instruction is broken up into its constituents. For example, the x86 instruction PUSH [EAX], which pushes onto the stack the value pointed to by register EAX, is compiled into

$$\mathsf{LOAD}\ \mathsf{TMP}_1,\ [\mathsf{EAX}]; \mathsf{STORE}\ [\mathsf{ESP} - 4],\ \mathsf{TMP}_1; \mathsf{SET}\ \mathsf{ESP},\ \mathsf{ESP} - 4$$

It's worth pointing out that, while the temporary registers introduced here bear a superficial resemblance to the infinite supplies of temporaries typically associated with compiler intermediate languages, they are purely a semantic trick and need no accounting for how they may be supported "on real hardware." They only appear in the language to allow us to decompose x86 instructions as in the example above, since there aren't, in general, any spare "real" registers to take the place of, e.g., TMP_1 in the example.

The details of SAL and Reduction are not especially enlightening, so I omit them here. The main technical component is the expected compatibility relation between states of the two kinds of programs, along with a compilation function and a proof that it respects compatibility.

5.3.3 FixedCode

The most basic knowledge a model checker needs is how to determine which instructions are executed when. The full semantics of SAL programs allows writing to arbitrary parts of memory, including those thought of as housing the program. We usually don't want to allow for this possibility and would rather simplify the verification framework. The FixedCode module is used to build verifiers based on this assumption. It is a functor that takes as input an abstraction that assumes a fixed code segment and returns an abstraction that is sound for the true semantics.

We modify the verifier signature used by ModelCheck. (Recall the convention introduced in Section 5.3.1 that definitions that we present in the context of a component should be treated as amending the input signature of the previous component.) The first addition is a memory value *prog* that contains in some contiguous region of its address space the encoding of the fixed program. The memory region *code* tells us which address space range this is.

$$\begin{array}{rcl} prog & : & \mathsf{word} \rightarrow \mathsf{byte} \\ code & : & \mathsf{memoryRegion} \end{array}$$

Next, we have a function *absPc* for determining the program counter for some subset of the abstract states. The model checker that FixedCode outputs will take responsibility for determining which instruction is next to execute in any state for which *absPc* returns SSome of a program counter. Other states may only be used as hypotheses and never appear directly in the abstract state space. For instance, we don't know the precise program counter of a hypothesis describing a return pointer, but this doesn't matter, since we are sure to visit all of the concrete program locations it could stand for.

$$\begin{array}{rcl} absPc & : & \forall \alpha : absState.\ \{\!\!\{ pc : \mathsf{word} \\ & & \quad \mid \forall \Gamma.\ \forall s.\ \Gamma \vdash_S s : \alpha \rightarrow s.pc = pc \}\!\!\} \end{array}$$

The final change to the signature of an abstraction is that, of course, we must now require that the code is never overwritten. If it were, then we would no longer know at verification time which instruction was being executed when, since the verifier will simply look instructions up in *prog* for this purpose. The progress condition of *step* is augmented to require that the destination of any STORE instruction is outside of the code region, using an auxiliary function instrOk.

$$\begin{array}{rcl} \mathsf{progress}(\alpha) & = & \forall s : Mac.state.\ \forall \Gamma : context. \\ & & \quad \Gamma \vdash_S s : \alpha \rightarrow \exists s' : Mac.state, s \mapsto_{Mac} s' \\ & & \quad \wedge\ \underline{\mathsf{instrOk}(s)} \end{array}$$

To define instrOk, I use the notation $|e|_s$ to stand for the result of evaluating expression e in machine state s, and the operation $\mathsf{lookupInstr}(s)$ decodes the instruction pointed to by s's program counter in s's memory.

$$\mathsf{instrOk}(s) \quad = \quad \begin{cases} |dst|_s \notin code, & \mathsf{lookupInstr}(s) = \mathsf{STORE}\ [dst],\ src \\ \mathsf{True}, & \text{otherwise} \end{cases}$$

5.3.4 TypeSystem

The next stage in the pipeline is the first where a significant decision is made on structuring verifiers. The TypeSystem component provides support for a standard approach to structuring abstract state descriptions: considering the value of each register separately by describing it with a type. I give an overview here of the signature required of type systems. I omit many of the details, but the pieces I've chosen to include illustrate the key points.

The basic idea is that we have an abstraction as before that can assume that, in addition to the custom abstract state that it maintains itself, a type assignment to each register is available at each step. The abstraction provides the set *ty* of types, along with a typing relation $\vdash_T$ to define their meanings, plus a subtyping procedure $\leq_T$.

$$\begin{array}{rcl} ty & : & \mathsf{Set} \\ \vdash_T & : & \mathit{context} \rightarrow \mathsf{word} \rightarrow ty \rightarrow \mathsf{Prop} \\ \leq_T & : & \forall \tau_1 : ty.\ \forall \tau_2 : ty.\ [\![\forall \Gamma.\ \forall w. \\ & & \quad \Gamma \vdash_T w : \tau_1 \rightarrow \Gamma \vdash_T w : \tau_2]\!] \end{array}$$

It's worth noting that there is no need to go into further detail on exactly how to allow typing relations to be defined. Coq's very expressive logic is designed for just such tasks, and natural-deduction style definitions of type systems via inference rules are accommodated naturally by the same mechanism for inductive type definitions that I demonstrated in Section 4.1, where the defined relation is placed in sort Prop.

We can see how our running example fits this signature, setting $ty = \mathsf{type}$. We first expand our notion of contexts to include a map from memory locations to types, in addition to the return pointer it stored before:

$$\mathit{context} \quad = \quad \mathsf{word} \times (\mathsf{word} \rightarrow \mathsf{type})$$

Now a simple set of typing rules suffices:

$$\frac{}{\Gamma \vdash_T w : \mathsf{int}} \quad \frac{\Gamma(w) = \tau}{\Gamma \vdash_T w : \tau \ \mathsf{ptr}} \quad \frac{}{\Gamma \vdash_T \Gamma.\mathit{retptr} : \mathsf{retptr}}$$

$\Gamma(w)$ denotes looking up the type of a memory location in the type map part of Γ, and $\Gamma.\mathit{retptr}$ denotes projecting out Γ's return pointer. We define the state abstraction relation $\vdash_S$ to enforce that every memory location really does have the type assigned to it. Here we are following the syntactic approach to FPCC [40] by using these type maps in place of potentially-complicated recursive conditions in typing rules.

We can also define a simple subtyping relation:

$$\frac{}{\tau \leq_T \mathsf{int}} \quad \frac{}{\tau \leq_T \tau}$$

This is "semantic" subtyping in the sense that we are identifying a (possibly strict) subset of the pairs of types that can be proved compatible in terms of $\vdash_T$, rather than defining a new syntactic notion.

Naturally, TypeSystem will need a way to determine the types of expressions if it is to track the register information that an abstraction assumes is available. The *typeof* function that a client abstraction must provide explains how to do this.

$$\begin{array}{lll} \mathit{typeof} & : & \forall \alpha : \mathit{absState}.\ \forall \vec{r} : \mathsf{reg} \to \mathit{ty}.\ \forall e : \mathsf{exp}. \\ & & \{\tau : \mathit{ty} \mid \forall \Gamma.\ \forall s. \\ & & \quad \Gamma \vdash_S s : \alpha \\ & & \quad \to (\forall r.\ \Gamma \vdash_T s.\mathsf{regs}(r) : \vec{r}(r)) \\ & & \quad \to \Gamma \vdash_T |e|_s : \tau\} \end{array}$$

Given an abstract state α, a register type assignment $\vec{r}$, and an expression e, *typeof* must return a type that describes the value of the expression in any compatible context Γ and concrete state s. It only needs to work correctly under the assumption that, in Γ, α

accurately describes s and $\vec{r}$ accurately describes all of s's register values.

typeof is quite uninteresting for our running example. Omitting proof components, we have:

$$\mathit{typeof}(\alpha, \vec{r}, e) \quad = \quad \begin{cases} \vec{r}(r), & e = \text{register } r \\ \mathsf{int}, & \text{otherwise} \end{cases}$$

As SAL expressions are side effect-free, we need a separate *typeofLoad* function for a memory dereference.

$$\begin{aligned} \mathit{typeofLoad} \quad : \quad & \forall \alpha : \mathit{absState}.\ \forall \vec{r} : \mathsf{reg} \to \mathit{ty}.\ \forall e : \mathsf{exp}. \\ & \{\!\{\tau : \mathit{ty} \mid \forall \Gamma.\ \forall s. \\ & \quad \Gamma \vdash_S s : \alpha \\ & \quad \to (\forall r.\ \Gamma \vdash_T s.\mathsf{regs}(r) : \vec{r}(r)) \\ & \quad \to \Gamma \vdash_T s(|e|_s) : \tau\}\!\} \end{aligned}$$

Note that *typeofLoad*'s range is a partial subset type, since not all types are valid for reading. For our running example, we have the following, using $\perp$ to denote `SNone`, the failure case:

$$\mathit{typeofLoad}(\alpha, \vec{r}, e) \quad = \quad \begin{cases} \tau, & \mathit{typeof}(\alpha, \vec{r}, e) = \tau\ \mathsf{ptr} \\ \perp, & \text{otherwise} \end{cases}$$

The full TypeSystem signature also includes an analogue for writes, enforcing that a "writable pointer" can't point into program memory and other similar conditions.

viewShift provides an important piece of logic that might not be obvious by analogy from type systems for higher-level languages. At certain points in its execution (and so in model checking), a program "crosses an abstraction boundary" which takes a different view of the types of values. A canonical example is a function call. The stack pointer register may switch from type "pointer to the fifth stack slot in my frame" to "pointer to the first stack slot in my frame." In the presence of type polymorphism via type variables, a register's type may change from "pointer to integer" to "pointer to β," where β is a type variable instantiated to "integer" for the call. There are many ways of structuring modularity in programs, so it's important that the requirements on *viewShift* be very flexible. Its type is:

$$
\begin{array}{lll}
\mathit{viewShift} & : & \forall \alpha : \mathit{absState}.\ \forall \vec{r} : \mathsf{reg} \to \mathit{ty}.\ \forall i : \mathsf{instr}. \\
& & \{\!\{(\alpha', \vec{r'}) : \mathit{absState} \times (\mathsf{reg} \to \mathit{ty}) \\
& & \quad \mid \forall \Gamma.\ \forall s. \\
& & \quad\ \ \Gamma \vdash_S s : \alpha \\
& & \quad\ \ \to (\forall r.\ \Gamma \vdash_T s.\mathsf{regs}(r) : \vec{r}(r)) \\
& & \quad\ \ \to \exists \Gamma', \Gamma' \vdash_S s : \alpha' \\
& & \quad\ \ \ \ \wedge (\forall r.\ \Gamma' \vdash_T s.\mathsf{regs}(r) : \vec{r'}(r))\}\!\}
\end{array}
$$

The type expresses that *viewShift* may provide any new abstract state α' and register type assignment $\vec{r'}$ for which there exists some context Γ' in which α' and $\vec{r'}$ are correct whenever α and $\vec{r}$ were correct in the original Γ. Another way of thinking of *viewShift* is that is a hook into the generic verifier that makes it possible to "cast" abstract states whenever a proof can be provided that the cast's target state is "no narrower than" the source state.

To illustrate a simple use of *viewShift*, we extend our running example type system temporarily with parametric polymorphism:

$$
\begin{array}{rrcl}
\mathsf{typevar} & \beta & & \\
\mathsf{type} & \tau & ::= & \mathsf{int} \mid \tau\ \mathsf{ptr} \mid \mathsf{retptr} \mid \beta \\
& \mathit{context} & = & \mathsf{word} \times (\mathsf{word} \to \mathsf{type}) \times (\mathsf{typevar} \to \mathsf{type})
\end{array}
$$

We might specify the entry point to a polymorphic identity function like this:

$$((39, [\mathsf{EAX} \mapsto \beta, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \beta])\})$$

If the concrete return pointer for a particular call is 92, then the state before calling the function for type int might be:

$$((88, [\mathsf{EAX} \mapsto \mathsf{int}]), \{(\mathsf{retptr}, [])\})$$

Incorporating the effects of the call instruction, *viewShift* returns the abstract state for the function's entry point. Starting from a context (pc, w, v), we construct a new context $(92, w, [\beta \mapsto \mathsf{int}])$ to justify the equivalence of the old and new abstract states.

It's worth recalling the setting in which TypeSystem is being used, which is to support construction of Coq terms to be extracted to OCaml code. $\vdash_T$ exists only in the Prop world, and so it will not survive the extraction process; it is only important in the proof of correctness of the resulting verifier. The types in *ty* are manipulated explicitly at verification time, so those survive extraction intact. $\leq_T$, *typeof*, and *viewShift* have both computational and logical content. For instance, the extracted version of $\leq_T$ is a potentially incomplete decision procedure with boolean answers. The extracted OCaml version of the verifier ends up looking like a standard type checker. You can think of the Coq implementation as combining a type checker and a proof of soundness for the type system it uses. There is considerable practical benefit from developing both pieces in parallel through the use of dependent types.

5.3.5 StackTypes

There are a wide variety of interesting type systems worth exploring for verifying different kinds of programs. At least when using the standard x86 calling conventions, every one of these type systems needs to worry about keeping track of the types of stack slots, which registers point to which places in the stack, proper handling of callee-save registers, and other such annoyances. StackTypes handles all of these details by providing a functor from a stack-ignorant TypeSystem abstraction to a TypeSystem abstraction aware of stack and calling conventions. The input abstraction can focus on the interesting aspects of the new types that it introduces rather than getting bogged down in the details of stack and calling conventions.

To make this feasible, the input abstraction only needs to provide a few new elements. First, a region of memory is designated to contain the runtime stack. It is accompanied with a proof that it has no overlap with the region where the program is stored.

$$\begin{array}{rcl} \mathit{stack} & : & \mathsf{memoryRegion} \\ \mathit{stackCodeDisjoint} & : & \mathsf{disjoint}(\mathit{stack}, \mathit{code}) \end{array}$$

The remaining ingredient is a way of making sure that the custom verification code of the abstraction will never allow the stack to be overwritten. The *checkStore* function is used for this purpose, being called on an expression that is the target of a STORE instruction to make sure that it won't evaluate to an address in the stack region.

$$\begin{array}{rcl} \mathit{checkStore} & : & \forall \alpha : \mathit{absState}.\ \forall \vec{r} : \mathsf{reg} \to \mathit{ty}.\ \forall e : \mathsf{exp}. \\ & & \llbracket \forall \Gamma.\ \forall s.\ \Gamma \vdash_S s : \alpha \\ & & \quad \to (\forall r.\ \Gamma \vdash_T s.\mathsf{regs}(r) : \vec{r}(r)) \\ & & \quad \to |e|_s \notin \mathit{stack} \rrbracket \end{array}$$

In my current implementation, the type of *checkStore* "exposes" the underlying stack implementation, though the client of StackTypes can avoid worrying too much about these details through the use of a library of helper functions.

With these ingredients, StackTypes builds a verifier that adds a few new types to the input abstraction's set *ty*. First, there are types Stack_i, indicating the ith stack slot from the beginning of the stack frame. Types for the stack slots are tracked in another part of abstract states. With this additional information, it's possible to determine the type of the value lying at a certain offset from the address stored in a register of Stack_i type. There is also a type Saved_r for each callee-save register r, denoting the initial value of r on entry to the current function call. These values will probably be saved in stack slots, and we will require that each callee-save register again has its associated Saved type when we return from the function. We know that we've reached this point when we do an indirect jump to a value of type Retptr, where the saved return pointer on the stack is given this type at the entry point to the function. While we've included an ad-hoc retptr type in our example so far, from here on we'll let StackTypes add it uniformly to whatever type system we're considering.

Perhaps surprisingly, StackTypes was the most involved to construct out of the components that I've listed. The most complicated implementation work dealt with the view shifts that occur during function calls and returns. Precise, bit-level proofs about calling conventions end up much more complex than they seem from an informal understanding. There is definitely room here for better automatic decision procedures for the theory of fixed-precision integers, as those sorts of proofs made up a good portion of the work.

We can now look at our running example, minus the ad-hoc retptr type, run through the StackTypes functor. While we have been using a simple fictitious calling convention to this point, we are ready now for a more realistic example. Here's a reasonable state description to apply right after the preamble at the start of a function from int to int ptr. The general organization is reminiscent of stack-based typed assembly language [65].

$$((81, [\mathsf{ESP} \mapsto \mathsf{Stack}_8], [0 \mapsto \mathsf{int}, 4 \mapsto \mathsf{Retptr}, 8 \mapsto \mathsf{Saved}_{\mathsf{EBX}}]),$$
$$\{(\mathsf{Retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{Saved}_{\mathsf{EBX}}], [])\})$$

We follow the x86 C calling convention, though I omit some of the details here for clarity. On the stack, we have (in order) the function's argument, the saved return pointer, and the saved value of register EBX (which is designated as callee-save). The stack pointer register ESP points to the end of these values. The return hypothesis tells us that, before returning, the function must have stored the int ptr return value in EAX and restored EBX to its original value.

To allow us to define *checkStore*, we need to modify our original typing rules to preclude ptr-type values that point into the stack:

$$\frac{\Gamma(w) = \tau \quad w \notin \mathit{stack}}{\Gamma \vdash_T w : \tau\ \mathsf{ptr}}$$

We as clients of StackTypes don't need to define typing rules for the stack-related type constructors, because StackTypes adds those to our typing judgment parametrically. The domain of contexts is expanded similarly to include the concrete starting address of the current stack frame and a map from callee-save registers to their initial values. The custom abstract states are extended to record stack frame length information.

5.3.6 SimpleFlags

In x86 machine language, there are no instructions that implement conditional test and jump atomically. Instead, all arithmetic operations set a group of flag registers, such as Z, to indicate a result of zero; or C, to indicate that a carry occurred. Each condition, formed from a flag and a boolean value, has a corresponding conditional jump instruction that jumps to a fixed code location iff that condition is true relative to the current flag settings. Thus, to properly determine what consequences follow from the fact that a conditional jump goes a certain way, it's necessary to track the relationship of the flags to the other aspects of machine states. Understanding these jumps is critical for such purposes as tracking pointer nullness and array bounds checks.

SimpleFlags is a functor that does the hard part of this tracking for an arbitrary abstraction, feeding its results back to the abstraction through this function:

$$
\begin{array}{lcl}
\mathit{considerTest} & : & \forall \alpha : \mathit{absState}.\ \forall \vec{r} : \mathsf{reg} \to \mathit{ty}.\ \forall co : \mathsf{cond}. \\
& & \quad \forall \circ : \mathsf{binop}.\ \forall e1, e2 : \mathsf{exp}.\ \forall b : \mathsf{bool}. \\
& & \quad \{\!\{(\alpha', \vec{r'}) : \mathit{absState} \times (\mathsf{reg} \to \mathit{ty}) \\
& & \quad\quad \mid \forall \Gamma.\ \forall s.\ \Gamma \vdash_S s : \alpha \\
& & \quad\quad\quad \to (\forall r.\ \Gamma \vdash_T s.\mathsf{regs}(r) : \vec{r}(r)) \\
& & \quad\quad\quad \to |e_1 \circ e_2|_s^{co} = b \\
& & \quad\quad\quad \to \exists \Gamma', \Gamma' \vdash_S s : \alpha' \\
& & \quad\quad\quad\quad \wedge (\forall r.\ \Gamma' \vdash_T s.\mathsf{regs}(r) : \vec{r'}(r))\}\!\}
\end{array}
$$

The type of *considerTest* looks similar to the type of *viewShift* from TypeSystem. Its purpose is to update an abstract state to reflect the information that a particular condition is true. The arguments α and $\vec{r}$ are as for *viewShift*. co names one of the finite set of conditions that can be tested with conditional jumps. $\circ$, $e1$, and $e2$ describe the arithmetic operation that was responsible for the current status of co. Finally, b gives the boolean value of co for this operation, determined from the result of a conditional jump. The notation $|e_1 \circ e_2|_s^{co}$ denotes the value of co resulting from evaluating the arithmetic operation $e_1 \circ e_2$ in state s.

Behind the scenes, SimpleFlags works by maintaining in each abstract state a partial map from flags to arithmetic expressions. The presence of a mapping from flag f to $e_1 \circ e_2$ means that it is known for sure that the value of f comes from $e_1 \circ e_2$, as it would be evaluated in the current state. SimpleFlags must be careful to invalidate a mapping conservatively each time a register that appears in it is modified. At each conditional jump, SimpleFlags checks to see if the relevant condition's value is known based on the flag map. If so, it calls *considerTest* to form each of the two abstract successor states, corresponding to the truth and falsehood of the condition.

We can demonstrate the basic operation with another extension of our running example. We consider the simplified setting where the only flags are Z (zero) and C (carry). For clarity, we ignore the StackTypes functionality added in the last subsection and focus instead on registers alone.

We extend our type system with nullness information on pointers:

$$\mathsf{type} \quad \tau \quad ::= \quad \mathsf{int} \mid \tau \ \mathsf{ptr} \mid \underline{\tau \ \mathsf{ptr}^?} \mid \mathsf{retptr}$$

$\tau \ \mathsf{ptr}^?$ is the new type standing for a possibly-null pointer to τ.

Now consider the following program state.

$$((81, [\mathsf{EAX} \mapsto \mathsf{int} \ \mathsf{ptr}^?], []), \{(Retptr, [], [])\})$$

We omit stack slot type information and include flag information in its place. Here, EAX is a possibly-null pointer to int, and nothing is known about the values of the flags.

If the next instruction is $\mathsf{CMP}\ 0,\ \mathsf{EAX}$, which compares the value in EAX against the constant 0 to set the flags, we would transition to a state like:

$$((85, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}^?], [Z \mapsto \mathsf{EAX} - 0, C \mapsto \mathsf{EAX} - 0]), \{(\mathit{Retptr}, [], [])\})$$

If after this we have a $\mathsf{JCC}\ Z,\ 123$ instruction, two successor states are produced. When the Z flag is true, we get the successor:

$$((123, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}^?], [Z \mapsto \mathsf{EAX} - 0, C \mapsto \mathsf{EAX} - 0]), \{(\mathit{Retptr}, [], [])\})$$

When Z is false, we get:

$$((89, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}], [Z \mapsto \mathsf{EAX} - 0, C \mapsto \mathsf{EAX} - 0]), \{(\mathit{Retptr}, [], [])\})$$

While I describe the choice here as though the program verifier determines which branch is taken, it actually conservatively queues both successors for exploration.

How were these successors generated? The *considerTest* function provided by our running example must recognize the special case where $co = (\mathsf{Z}, \mathsf{true})$, $\circ = -$, e_1 is some register r, $e_2 = 0$, $b = \mathsf{false}$, and $\vec{r}(r) = \tau\ \mathsf{ptr}^?$ for some τ. In other words, we have verified that some register of type $\tau\ \mathsf{ptr}^?$ is nonzero. In such a case, we promote r to type $\tau\ \mathsf{ptr}$ in the returned $\vec{r'}$.

If the instruction at location 123 is $\mathsf{MOV\ EAX},\ 3$, which stores the constant 3 in register EAX, then the state that will result is:

$$((123, [\mathsf{EAX} \mapsto \mathsf{int}], []), \{(\mathit{Retptr}, [], [])\})$$

The custom code for our example is not involved in this transition. Instead, SimpleFlags sees that the value of a register mentioned in all of the known flag states has changed, and so it erases all of that flag information.

After reading these last two subsections, the reader may be wondering why StackTypes and SimpleFlags received this relative order. Indeed, neither depends on the other, and this is the one case in the component pipeline where a reordering would have been acceptable.

5.3.7 WeakUpdate

We've now built up enough machinery to get down to the interesting part of a type-based verifier, designing the type system. WeakUpdate provides a functor for building verifiers from type systems of a particular common kind. These are type systems that are based on *weak update* of memory locations, where each accessible memory cell has an associated type that doesn't change during the course of a program run. A cell may only be overwritten with a value of its assigned type. Of course, with realistic language implementations, storage will be reused, perhaps after being reclaimed by a garbage collector. Though handling storage reclamation is beyond the scope of this work, I believe that the proper approach is to verify each program with respect to an abstract semantics where storage is never reclaimed, separately verify a garbage collector in terms of the true semantics, and combine the results via a suitable composition theorem.

I'll now present the signature of a WeakUpdate type system piece by piece. In contrast to the signatures given for the previous components, this signature does not extend its predecessors. With one small exception that I will describe below, the elements that

I'll list are all that a type system designer needs to provide to produce a working verifier with a proof of soundness. It's also true that, while I've simplified the presentation of the signatures in previous subsections, this signature is a literal transcription of most of the requirements imposed by the real implementation.

Like for the TypeSystem module, a WeakUpdate type system is based around a set ty of types, with a typing relation $\vdash_T$ and a subtyping procedure $\leq_T$. An important difference is that here we hard-code contexts to be partial maps from memory addresses to types. Other standard context elements from previous examples, such as the saved return pointer, will now be handled internally by the WeakUpdate functor. As WeakUpdate is only intended as one simple example of a terminal verification component, it doesn't support polymorphism in the style found in some earlier examples.

$$
\begin{array}{rcl}
ty & : & \mathsf{Set} \\
context & = & \mathsf{word} \rightarrow \mathsf{option}\ ty \\
\vdash_T & : & context \rightarrow \mathsf{word} \rightarrow ty \rightarrow \mathsf{Prop} \\
\leq_T & : & \forall \tau_1 : ty.\ \forall \tau_2 : ty.\ [\![\forall \Gamma.\ \forall w. \\
& & \quad \Gamma \vdash_T w : \tau_1 \rightarrow \Gamma \vdash_T w : \tau_2]\!]
\end{array}
$$

A few simple procedures suffice to plug into a generic type-checker for machine code:

$$
\begin{aligned}
\mathit{typeofConst} \quad &: \quad \forall w : \mathsf{word}.\ \{\tau : \mathit{ty} \mid \forall \Gamma.\ \Gamma \vdash_T w : \tau\} \\[1em]
\mathit{typeofArith} \quad &: \quad \forall \circ : \mathsf{binop}.\ \forall \tau_1, \tau_2 : \mathit{ty}.\ \{\tau : \mathit{ty} \\
&\qquad \mid\ \forall \Gamma.\ \forall w_1.\ \forall w_2.\ \Gamma \vdash_T w_1 : \tau_1 \\
&\qquad\quad \to\ \Gamma \vdash_T w_2 : \tau_2 \\
&\qquad\quad \to\ \Gamma \vdash_T w_1 \circ w_2 : \tau\} \\[1em]
\mathit{typeofCell} \quad &: \quad \forall \tau : \mathit{ty}.\ \{\!\{\tau' : \mathit{ty} \mid\ \forall \Gamma.\ \forall w. \\
&\qquad \Gamma \vdash_T w : \tau \\
&\qquad \to\ \Gamma(w) = \mathsf{Some}\ \tau'\}\!\}
\end{aligned}
$$

typeofConst gives a type for every constant machine word value; *typeofArith* gives a formula for calculating the type of an arithmetic operation in terms of the types of its operands; and *typeofCell* provides a function from a pointer type to the type of any values that it may point to, returning SNone for non-pointer types.

The final element is a way of taking advantage of knowledge of conditional jump results, based behind the scenes on SimpleFlags:

$$
\begin{aligned}
\mathit{considerNeq} \quad &: \quad \forall \tau : \mathit{ty}.\ \forall w : \mathsf{word}.\ \{\tau' : \mathit{ty} \mid\ \forall \Gamma.\ \forall w'. \\
&\qquad \Gamma \vdash_T w' : \tau \\
&\qquad \to\ w' \neq w \to \Gamma \vdash_T w' : \tau'\}
\end{aligned}
$$

When the result of a conditional jump implies that some value of type τ is definitely not equal to a word w, *considerNeq* is called with τ and w to update the type of that value to reflect this. A canonical example of use of *considerNeq* is with a nullness check on a pointer, to upgrade its type from "pointer" to "non-null pointer."

The two significant omissions from this signature description are functions very similar to *considerNeq*. They consider the cases where not a value itself but *the value it* *points to in memory* is determined to be equal to or not equal to a constant. A canonical example of usage of these functions is in compilation of case analysis over algebraic datatypes.

The proper use by WeakUpdate of these three functions requires some quite nontrivial bookkeeping. WeakUpdate performs a very simple kind of online points-to analysis to keep up-to-date on which values particular tests provide information on. The most complicated relationship tracked by the current implementation is one such as: TMP_1 holds the result of dereferencing EAX, which holds a value read from stack slot 6. If stack slot 6 is associated with a local variable of a sum type, then a comparison of TMP_1 with some potential sum tag should be used to update the types of both EAX and stack slot 6 to rule out some branches of the sum. As for SimpleFlags, WeakUpdate must be careful to erase a saved relationship when it can't be sure that a modification to a register or to memory preserves it.

Happily, these complications need not concern a client of WeakUpdate. In the next section, I illustrate this with a simple use of it to construct a type system handling some standard types for describing linked, heap-allocated structures.

First, I'll give a quick overview of how our running example can be expressed succinctly as a WeakUpdate type system, ignoring proof components as in previous sections. We add a singleton integer type to our type system, for reasons that should become clear shortly.

$$\mathsf{type} \quad \tau \quad ::= \quad \mathsf{int} \mid \tau\ \mathsf{ptr} \mid \tau\ \mathsf{ptr}^? \mid \underline{\mathsf{S}(n)}$$

We use these typing rules:

$$\frac{}{\Gamma \vdash_T w : \mathsf{int}} \quad \frac{}{\Gamma \vdash_T w : \mathsf{S}(w)}$$

$$\frac{\Gamma(w) = \tau \quad w \notin \mathit{code} \cup \mathit{stack}}{\Gamma \vdash_T w : \tau\ \mathsf{ptr}} \quad \frac{}{\Gamma \vdash_T 0 : \tau\ \mathsf{ptr}^?} \quad \frac{\Gamma(w) = \tau \quad w \notin \mathit{code} \cup \mathit{stack}}{\Gamma \vdash_T w : \tau\ \mathsf{ptr}^?}$$

We can use this simple subtyping relation:

$$\frac{}{\tau \leq_T \mathsf{int}} \quad \frac{}{\tau \leq_T \tau} \quad \frac{}{\mathsf{S}(0) \leq_T \tau\ \mathsf{ptr}^?} \quad \frac{}{\tau\ \mathsf{ptr} \leq_T \tau\ \mathsf{ptr}^?}$$

The pieces we need to plug into the extensible type-checker are:

$$\begin{array}{rcl}
\mathit{typeofConst}(w) & = & \mathsf{S}(w) \\
\mathit{typeofArith}(\circ, \tau_1, \tau_2) & = & \mathsf{int} \\
\mathit{typeofCell}(\tau) & = & \begin{cases} \tau', & \tau = \tau'\ \mathsf{ptr} \\ \bot, & \text{otherwise} \end{cases} \\
\mathit{considerNeq}(\tau, w) & = & \begin{cases} \tau'\ \mathsf{ptr}, & \tau = \tau'\ \mathsf{ptr}^? \text{ and } w = 0 \\ \tau, & \text{otherwise} \end{cases}
\end{array}$$

...and that's about the entirety of a formal description of this verifier. I've omitted proofs in this summary, but an actual implementation of this verifier (included as an example with the software distribution) is only about 150 lines of Coq code, most of it concerned with filling in trivial proofs for default implementations of the various operations. The end result is a certified verifier that runs on real x86 binaries.

5.3.8 Architecture Summary

I review the overall pipeline by summarizing it again, this time in top-down order:

1. **WeakUpdate**: High-level type system description
2. **SimpleFlags**: Add instrumentation tracking correlations between conditional flags and register values.

3. **StackTypes**: Add standard types modeling stack and calling conventions.

4. **TypeSystem**: Add tracking of register types by plugging user-specified handlers into a generic type checker.

5. **FixedCode**: Add enforcement of immutability of the program code area in memory.

6. **Reduction**: Convert from a verifier for a simple RISC language to a verifier for x86 assembly language.

7. **ModelCheck**: Implement the actual state-space traversal.

The input to any stage involves both algorithmic pieces and proof pieces. A client of the library needn't concern himself with the interfaces of components coming later in this list than the one he chooses to use as a starting point. Each component assembles new executable verifier code and new soundness proofs from those passed to it as input, with the composition of the components in any suffix of this list spanning the entire corresponding abstraction gap.

5.4 Case Study: A Verifier for Algebraic Datatypes

In this section, I will present some highlights of the Coq implementation of the MemoryTypes verifier, based on the library components from the last section. Recall that this verifier is designed to handle programs that use algebraic datatypes. For clarity, I've made some simplifications from the real Coq code, especially regarding dependent pattern matching.

MemoryTypes is implemented simply using WeakUpdate, the last component described in the last section. All we need to do is provide an implementation of WeakUpdate's input signature, which I described in Section 5.3.7. The first step is to define the language of types, which includes the standard low-level types used to implement algebraic datatypes.

```
Definition var := nat.

Inductive ty : Set :=
  | Constant : int32 -> ty
  | Product : list ty -> ty
  | Sum : ty -> ty -> ty
  | Var : var -> ty
  | Recursive : var -> ty -> ty.
```

You can see that the set `ty` of types includes the elements you would expect; namely, constructors for building product, sum, and recursive types in the usual ways. There are also `Constant` types for sum tags of known values and `Var` types to represent the bound variables of recursive types.

Next we must define the typing relation:

```
Inductive hasTy : context -> int32 -> ty -> Prop :=
  | HT_Constant : forall ctx v,
    hasTy ctx v (Constant v)
  | HT_Unit : forall ctx v,
    hasTy ctx v (Product nil)
  | HT_Product : forall ctx v t ts,
    ctx v = Some t
    -> hasTy ctx (v + 4) (Product ts)
    -> hasTy ctx v (Product (t :: ts))
  | HT_Suml : forall ctx v t1 t2,
    hasTy ctx v (Product (Constant 0 :: t1 :: nil))
    -> hasTy ctx v (Sum t1 t2)
  | HT_Sumr : forall ctx v t1 t2,
    hasTy ctx v (Product (Constant 1 :: t2 :: nil))
    -> hasTy ctx v (Sum t1 t2)
  | HT_Recursive : forall ctx x t v,
    hasTy ctx v (subst x (Recursive x t) t)
    -> hasTy ctx v (Recursive x t).
```

The typing relation `hasTy` is defined in terms of its inference rules in the standard way. You can see that the same inductive definition mechanism that is used for standard algebraic datatypes works just as naturally for defining judgments. We have that any word has the corresponding constant type; any word has the empty product type; a word has a non-empty product type if it points to a value with the first type in the product, and the following word in memory agrees with the remainder of the product; a word has a sum type if it has type $\texttt{Constant}(i) \times t$ where t corresponds to the ith element of the sum; and a word has a recursive type if it has the type obtained by unrolling the recursion one level.

Next we define the subtyping procedure:

```
Definition subTy : forall (t1 t2 : ty),
  [[forall ctx v, hasTy ctx v t1 -> hasTy ctx v t2]].
  intros t1 t2.
  refine (subTy' t1 t2
    || subTy' (tryUnrollingOnce t1) t2
    || subTy' t1 (tryUnrollingOnce t2)); ....
Qed.
```

The top-level procedure uses a subroutine `subTy'` to do most of the work. `subTy'` has no special handling of recursive types. Instead, `subTy` tries a heuristic set of possible unrollings of recursive types, calling `subTy'` on each result and concluding that the subtyping relation holds if any of these calls succeeds. It is the responsibility of a certifying compiler targeting this verifier to emit enough typing annotations that no more sophisticated subtyping relation is required, effectively splitting a multi-unrolling check into several simpler checks.

The definition of `subTy'` proceeds in the standard way, defining a recursive function with holes left to be filled in with tactic-based proof search:

```
Definition subTy' : forall (t1 t2 : ty),
  [[forall ctx v, hasTy ctx v t1 -> hasTy ctx v t2]].
  refine (fix subTy' (t1 t2 : ty) {struct t2}
```

```
    : [[forall ctx v,
      hasTy ctx v t1 -> hasTy ctx v t2]] :=
    match (t1, t2) with
      | ...
    end); ....
Qed.
```

The pattern matching cases have the expected implementations. For one example, consider the case for subtyping between constant types:

```
| (Constant n1, Constant n2) =>
  pfEq <- int32_eq n1 n2;
  Yes
```

Constant types are only compatible if their constants are equal. We use a richly-typed integer comparison procedure `int32_eq` with our monadic notation. The proof `pfEq` that results from a successful equality test will be used to discharge the proof obligation arising for the correctness of this case.

Another example case is that for comparing a product type of the proper form to a sum type:

```
| (Product (Constant n :: t :: nil), Sum t1 t2) =>
  (int32_eq n 0 && ty_eq t t1)
  || (int32_eq n 1 && ty_eq t t2)
```

A product type is only compatible with a sum type if the product starts with a constant tag identifying a branch of that sum and the next field of the product draws its type from the same sum branch. Here, the standard boolean operators are custom "macro" syntax defined to do proper threading of known facts through the expression to the points triggering proof obligations.

Omitted from this discussion are the `typeof*` functions and the `consider*` functions, which are used to update sum types based on conditional jump results. All of these work as you would expect, with nothing especially enlightening about their implementations. The other big omission is the specification of proof scripts, or sequences of tactics, that are required to describe strategies for proof construction. In many cases, these proof scripts are atomic calls to automating tactics, but in some cases they are longer than would be desired. Improving that aspect of verifier construction is a fruitful area for future work.

Nonetheless, the entire MemoryTypes implementation is only about 600 lines long. I was able to develop it in less than a day of work. Thanks to the common library infrastructure, the reward for this modest effort is a verifier with a rigorous soundness theorem with respect to the real bit-level semantics of the target machine.

5.5 Implementation

The source code and documentation for the system described in this chapter can be obtained from:

`http://proofos.sourceforge.net/`

The implementation is broken up into a number of separately-usable components: a library of Coq and OCaml code dealing with semantics and parsing of machine code in general and x86 machine code in particular; a Coq extension in support of extracting programs that use native integer types; and the certified verifiers library proper, including trusted code formalizing the problem setting and the collection of untrusted verification components highlighted in Section 5.3.

5.5.1 The Asm Library

When I began this project, I set out to survey the different choices of pre-packaged libraries formalizing useful subsets of x86 assembly language. I was quite surprised to find that, not only was there no such package available for my chosen proof assistant, but the situation seemed to be the same for all other choices of tools, as well! Since this kind of formalization is key to the project, I created my own, designing it as a library useful in its own right, called Asm.

Some of my choices veer away from formality and small trusted code bases in favor of practicality. The library deals with a small subset of x86 machine language that covers the instructions generated by GCC for the examples that I've tried. It defines in both Coq and OCaml a language of abstract syntax trees for machine code programs. There is a Coq operational semantics for these ASTs, along with an OCaml parser from real x86 binaries to ASTs. The extracted Coq verifiers use the OCaml parser as a "foreign function," sacrificing some "free" formal guarantees. For instance, the parser uses imperative code for file IO to initialize a global variable with program data. Coq's formal semantics contains no account of such phenomena, but I believe that no unsoundness is introduced because all OCaml "foreign functions" are observationally pure over single executions. In any case, some worthwhile future work would be to move more of the process into genuine Coq code.

The Asm library contains a few other related pieces that I wasn't able to find elsewhere, including formalizations of different fixed-width bitvector arithmetic operations and their properties. Altogether, the Coq code size tallies for Asm run to about 10,000 lines in a generic utility library, 1000 to formalize bitvectors and fixed-precision arithmetic, and

1000 to formalize a subset of x86 machine code.

5.5.2 Proof Accelerator

Different data structures make sense for rigorous formalization than for efficient execution. For instance, Coq's standard library defines natural numbers with the standard Peano-style recursive type, effectively representing them in unary. This provides a ready and effective induction principle, and issues of representation efficiency don't come into play when we're proving generic properties of the naturals and not examining particular individuals. However, when we extract to OCaml programs that use natural numbers and run them on particular inputs, we find ourselves in the unusual situation that basic arithmetic operations on natural numbers are a primary performance bottleneck. We want to *reason about* mathematical natural numbers but *represent them at run-time* in a way that takes advantage of built-in processor capabilities. Similar concerns apply for the fixed-width bitvector types that abound in any project working with machine language, though there we only hope for constant-factor performance improvements by using native types in our verifiers.

A Coq extension that I call "Proof Accelerator" achieves this by modifying the program extraction process in a very simple way. Custom type mappings are already supported by Coq, so I request that `nat` be mapped to the OCaml infinite-precision integer type (since the potential for silent overflow would invalidate formal proofs), the Coq type of 32-bit words to the native 32-bit word type, etc.. Using the same mechanism, we can identify some common functions (e.g., addition) that should be extracted to use their standard OCaml implementations. This leaves one more important class of modifications that

Coq doesn't support out of the box.

In particular, we need to rewrite uses of *pattern matching*. We have replaced inductive types with what are effectively abstract types, like native integer types. Proof Accelerator rewrites pattern matches using `when` guards and local bindings. The results should have the same semantics as the original.

The whole approach and implementation are quite informal. The practical effect of this implementation choice is that all of Proof Accelerator must be counted among the trusted code base, along with the Coq proof checker and extractor and the OCaml compiler. An interesting future project would be to look at a more general and rigorous system of proof-preserving transformations that replace one implementation of an abstract data type with another.

5.5.3 Certified Verifiers Library

I've already gone into detail in previous sections on the content of this piece. The different components combine to take up about 7000 lines of Coq code, highlighting the effectiveness of re-use, as only 700 lines were needed for the last section's case study, with about 150 of them either reasonable candidates for relocation to the general utility library or unavoidable boilerplate that wouldn't grow with verifier complexity.

Most of the resulting implementation is Coq code which is extracted to ML. For simplicity, I chose to implement in OCaml some pieces that must inevitably belong to the trusted code base, along the lines of the OCaml instruction decoding in the Asm library. There is also some OCaml code that has no effect on soundness; for instance, to read metadata from a binary and pass it to the extracted verifier as suggested preconditions for the basic blocks. Bugs in this metadata parsing can hurt completeness, but they can never lead to incorrect acceptance of an unsafe program. It would even be possible to replace this code with a complicated abstract interpreter that infers much of what is currently attached explicitly, and the results could be fed to the unchanged extracted verifier with the same soundness guarantees. (This, of course, is modulo the lack of formality that we accept when interfacing Coq and OCaml code.)

We can summarize the big picture of what these implementation pieces give us. A final verifier can be checked for soundness by running the Coq `Check` command on its entry point function and verifying that the type that is printed matches the $\forall p : \mathsf{program}.\ [\![\mathsf{safe}(\mathsf{load}(p))]\!]$ type I gave in Section 5.2.1. The "backwards slice" of definitions that this type depends on constitutes the trusted part of the development, and it contains only small parts of the Asm library and brief definitions of abstract machines and safety from the certified verifiers library.

5.6 Related Work

The verifiers produced in this project are used in the setting of proof-carrying code. Relative to our past work [13] on certified verifiers, my new contribution here is first, to suggest developing verifiers with Coq in the first place, instead of extracting verification conditions about more traditional programs; and second, to report on experience in the effective construction of such verifiers through the use of dependent types and re-usable components. Several projects [2, 40, 25] consider in a PCC setting proofs about machine code from first principles, but they focus on proof theoretical issues rather than the pragmatics of constructing proofs and verifying programs under realistic time constraints. Our certified verifiers approach allows the construction of verifiers with strong guarantees that nonetheless perform well enough for real deployment. Wu et al. [92] tackle the same problem based on logic programming, but they provide neither evidence of acceptable scalability of the results nor guidance on the effective engineering of verifiers as logic programs. We showed in our initial work on certified verifiers [13] that we can achieve verification times an order of magnitude better than both those of Wu et al. and those from our past work on the Open Verifier [14]. At the same time, the prototype verifier that we used for these measurements stayed within a factor of 2 of the running time of the traditional, uncertified Typed Assembly Language checker [64].

The architecture that I have presented is only a first step towards a general and practical system. Many Foundational PCC projects have considered in a proof theoretical setting ideas that could profitably be used in concert with certified verifiers. For instance, the component library that I've described only supports whole-program verification. In contrast, the progression of verification frameworks that culminates in OCAP [34] provides a rich setting for modular verification of systems whose pieces are certified using different program logics. It seems natural to consider the adaptation of this idea to cooperative verification using different certified verifiers. The technology of certified verifiers also has a good way left to go to "catch up" with the FPCC world by common metrics like sophistication of language features verified and minimality of the trusted code base.

Past projects have considered using proof assistants to develop executable abstract interpreters [12] and Java bytecode verifiers [51, 7]. My work differs in dealing with machine code, which justifies the kind of layered component approach that I've described, and my work focuses more on accommodating a wide variety of verification approaches without requiring the development of too much code irrelevant to the main new idea of a technique. My work has much in common with the CompCert project [57], which works towards a fully certified C compiler developed in Coq. The main differences are my use of dependent types to structure the "program" part of a development and my emphasis on reusable library components.

The Rhodium project [56] also deals with the construction of certified program analyses. By requiring that analyses be stated in a very limited Prolog-like language, Rhodium makes it possible to use an automated first-order theorem prover to discharge all proof obligations. This approach is very effective for the traditional compiler optimizations that the authors target, but it doesn't seem to scale to the kinds of analyses associated with FPCC. For instance, proofs about type systems like the one demonstrated here are subtle enough that human control of the proving process seems necessary, justifying our choice of a much richer analysis development environment.

The Epigram [59], ATS [15], and RSP [91] languages attempt to inject elements of the approach behind Coq program extraction into a more practical programming setting. I believe I have taken good advantage of many of Coq's mature features for proof organization and automation in ways that would have been significantly harder with these newer languages, which focus more on traditional programming features and their integration with novel proof manipulations. It's also true that much of the specifics of my approach to designing and implementing certified verifiers is just as interesting transposed to the contexts of those languages, and the ideas are of independent interest to the PCC community.

5.7 Conclusion

Based on the results I've reported here, I hope I've provided some evidence that technology that has been found in computer proof assistants for some time is actually already sufficient to support certified programming for non-toy problems. While recent proposals in this space focus on integrating proofs and dependent types with imperativity and other "impure" language features, I was able to construct a significant and reasonably efficient certified program verification tool without using such features. In other words, the advantages of pure functional programming are only amplified when applied in a setting based on rigorous logical proofs, and the strengths of functional programming and type theory are sufficient to support the construction of a program with a formal proof of a very detailed full correctness property.

Part II

Certified Compilers

Chapter 6

A Language Formalization Methodology

Most every researcher in programming languages has heard of computer tools for formalizing languages, their semantics, and tools that operate on them. Where pencil-and-paper methods have worked wonders for avoiding embarrassing errors in language design and implementation, computer-checked proofs and their relatives can only provide further assurance. Nonetheless, these formal tools are used largely only by people involved in researching their design. This leaves these researchers scratching their heads, wondering why everyone else is passing up this golden opportunity.

The simple reason is that formal tools like proof assistants are viewed as requiring an additional investment whose cost doesn't justify the new rewards. Learning to use a program like Coq, Isabelle, or Twelf is a journey in itself, and it's not clear that there is light at the end of the tunnel. Even the respective experts on these systems present formalizations that seem verbose and impenetrable to the novice. I would argue that it is not the case that these novices simply have more to learn before they can appreciate these examples. Rather, today's idiomatic techniques for mechanizing programming language metatheory are just too heavyweight.

The POPLmark Challenge [4] addresses these problems through proposing a series of formalization benchmarks and evaluating different solutions to them. The field of solutions submitted to the first challenge shows that we have some ground left to cover to obtain computerized proofs that can be regarded as transcriptions of reasonable pencil-and-paper proofs. We can't expect computer proofs to look as straightforward as their informal counterparts, since the originals' straightforwardness is usually deceiving and may even hide bugs, but this goal can be a useful "grand challenge" to work towards.

The techniques I present in this chapter are meant as a step in the right direction, and Chapters 7 and 8 follow with evidence for this based on a concrete compiler certification project. My approach can be viewed as an extension of a traditional progression towards more usable proof tools. Additionally, there is a twist that recommends a change to the way that we think about informal proofs, in consideration of how easily they can be formalized.

That traditional progression is based on a simply-stated principle: "Encode as many features as possible of your *object language* using closely related features of your *meta language*." Here, "object language" refers to the programming language or logic that you are formalizing, while "meta language" refers to the language of your chosen proof assistant program. We can step down the traditional elements of programming languages and see how this principle is applied to each one. Later on, I'll give concrete examples of each of these.

- *Concrete syntax.* No one formalizing programming languages expects to write a lexer or parser before he is able to get started on the interesting bits. All of the candidate proof assistants have parsers for a concrete input format sufficient for describing a wide variety of abstract syntaxes.

- *Abstract syntax.* When we get down to the essence of a high-level language's syntax, the biggest issue is usually representing variable binding constructs. We want somehow to enforce that each subterm uses only variables from some well-defined set based on lexical scoping or a similar concept. The technique of *higher-order abstract syntax (HOAS)* is used quite effectively to encode this in the logical frameworks community and elsewhere. The body of each object language binder is represented as a *meta language function from the value of the bound variable to the representation of the body.* In this way, the user avoids needing to formalize his own syntactic operations like substitution, since he inherits the meta language's support for them.

- *Static semantics*, including object-language type systems. While most proof tools allow separate formalization of abstract syntax trees (ASTs) and typing judgments, tools with dependently-typed meta languages support a simpler strategy. The abstract syntax trees can be defined as an *indexed type family* whose indices indicate what object language type a particular piece of syntax is meant to have. In this way, syntax and static semantics are combined, avoiding the need for typing judgments as side conditions throughout a development. One of the main advantages of this idea is that *transformations* on syntax can be verified to be type-preserving through relatively simple type-checking.

- *Dynamic semantics.* Here we get to the only aspect of a programming language that your average industry practitioner thinks about in any detail, so we had better have the best story for how to model it effectively. I would like to argue that, unfortunately, our techniques here actually score the worst out of the four categories. In both informal and computer-formalized proofs, operational semantics is today's standard technique. With operational semantics, we forego taking advantage of any direct correspondence between object and meta languages, opting instead to formalize everything from first principles in a very syntactic way. This may be very nice from a foundational theoretical perspective, but it's rather inconvenient for real engineering tasks.

To correct this last deficiency, I propose bringing back an old friend, *denotational semantics*. While largely out of style today thanks to difficulties formalizing some more advanced programming language features, denotational semantics is very effective when it works. Further, it has a very natural connection to the modeling pattern that I stated earlier: denotational semantics is based on the idea of reifying a translation from an object language to a meta language. Most of the proof tools out there aren't quite higher-order enough to allow this sort of thing to be represented in the meta language itself. There is one notable exception that I'm aware of among widely-used tools, the Coq proof assistant, which was one of my main reasons for choosing it for this thesis.

The rest of this chapter is dedicated to elaborating a simple two-step plan for more effective mechanized language metatheory:

1. Favor *denotational semantics* over *operational semantics* whenever possible.

2. Implement your denotational semantics in a way that *maps object language entities directly into meta language entities*, such that *the translation itself is a first-class entity of the meta language.*

The next section evolves a formalization of simply-typed lambda calculus through the standard techniques for representing variable binding, ending with my proposed representation technique and an exposition of its benefits for reasoning about dynamic semantics. Following that, I describe how my techniques allow effective reasoning about compilation, by way of an example where one lambda calculus is compiled into another and the compiler's correctness is proved very succinctly. More realistic examples are the domain of the following chapters, which present a certified compiler from lambda calculus to assembly language.

6.1 An Evolutionary Example: The Simply-Typed Lambda Calculus

I'll begin by stepping through a sequence of representation decisions for a formalization of the simply-typed lambda calculus. All are based on the type language represented by this Coq definition:

```
Inductive ty : Set :=
  | Base : ty
  | Arrow : ty → ty → ty
```

6.1.1 A Nominal Representation

The representation of terms that corresponds most closely to what we see in pencil-and-paper proofs is usually called a *nominal* representation.

$$\begin{array}{l}\mathsf{Inductive\ term : Set :=} \\ \quad \mid \mathsf{EVar : name} \rightarrow \mathsf{term} \\ \quad \mid \mathsf{Lam : name} \rightarrow \mathsf{term} \rightarrow \mathsf{term} \\ \quad \mid \mathsf{App : term} \rightarrow \mathsf{term} \rightarrow \mathsf{term}\end{array}$$

With this definition, the identity function on the base type that is written informally as $\lambda x : \mathsf{b}.\ x$ is translated to $\mathsf{Lam}\ x\ (\mathsf{EVar}\ x)$.

We have a type of *names* corresponding to the infinite domain of variable names assumed in informal proofs. A term is either a name, a lambda abstraction that binds a name in a term, or an application of two terms.

With this kind of representation, we need to apply operations like alpha-renaming manually to remain faithful to the informal version. Also, if we follow the standard route of defining an operational semantics for these terms, we will need to define a customized notion of capture-avoiding substitution. These extra requirements are unfortunate because informal proofs manage to elide them without too much trouble.

There has been some recent work on tool support for nominal syntax representations, including on the nominal datatypes package for Isabelle [87]. The user avoids much of the tedious work of thinking about alpha-renaming, but explicit side conditions about variable names still show up in places where we'd rather they didn't, and where they don't show up in informal proofs. Tool support of this kind also doesn't provide any help in effective definition of dynamic semantics. On that note, I move on to the next standard first-order representation technique.

6.1.2 De Bruijn Index Representation

This technique, that of *de Bruijn indices* [28], is also called a *nameless* representation. Instead of giving variables names, a variable is represented as a *natural number* addressing a binder in terms of how many binders "upward" it appears in lexical scoping order. As a result, binding constructs like function abstraction no longer need to name a variable.

$$\begin{array}{l} \mathsf{Inductive\ term} : \mathsf{Set} := \\ \quad \mid \mathsf{EVar} : \mathsf{nat} \to \mathsf{term} \\ \quad \mid \mathsf{Lam} : \mathsf{term} \to \mathsf{term} \\ \quad \mid \mathsf{App} : \mathsf{term} \to \mathsf{term} \to \mathsf{term} \end{array}$$

The base-type identity function is now written $\mathsf{Lam}\ (\mathsf{EVar}\ 0)$.

The nameless representation is nice because we avoid the need to state explicit side conditions about variable names. On the other hands, operations like bringing a new, unused variable into the scope of a de Bruijn term need to be implemented with explicit lifting functions, whereas they come "for free" for nominal terms. This awkwardness can be ameliorated partially through tool support along the lines of the Isabelle nominal datatypes package. I also want to argue that many of these awkward syntactic operations show up in formalizations based on operational semantics but are absent when using denotational semantics.

6.1.3 Higher-Order Abstract Syntax

The third basic binding representation technique is *higher-order abstract syntax (HOAS)* [75], which identifies object language binders with meta language binders. Here's an example pseudo-Coq definition that uses HOAS.

$$\begin{array}{l}\mathsf{Inductive\ term : Set :=} \\ \quad \mid \mathsf{Lam} : (\mathsf{term} \to \mathsf{term}) \to \mathsf{term} \\ \quad \mid \mathsf{App} : \mathsf{term} \to \mathsf{term} \to \mathsf{term}\end{array}$$

Our running example becomes $\mathsf{Lam}\ (\lambda x : \mathsf{term}.\ x)$.

We no longer have a constructor for variables. Instead, object language variables are represented with meta language variables. The abstraction constructor takes as its argument a *function* from terms to terms, building the body of the abstraction from a term that will in practice be a meta language variable.

Substitution and many other syntactic definitions and properties come for free with HOAS. However, writing *recursive functions* over HOAS terms is tricky. In fact, it's so tricky that Coq doesn't allow the normal variety of HOAS, which is why I called that last definition "pseudo-Coq." Without significant changes to other parts of its logic, Coq would become inconsistent were that definition allowed. Systems like Twelf have simpler meta languages where no such problems arise.

Some recent work [90] has proposed techniques based on modal logic for recursion on HOAS terms, but, in my opinion, these techniques are sufficiently more complicated than those associated with de Bruijn indices that they don't settle the matter. It's also true that these techniques are designed for meta languages that have been kept simple so that they have canonical forms properties that are important for applications in logical frameworks. The denotational semantics techniques that I will present here rely on having a much richer meta language, so techniques for HOAS recursion must return to the drawing board.

Nonetheless, I'd like to take this opportunity to dispel the myth that Coq forbids

HOAS altogether. The example I will give also sets the stage for the technique that I will settle on in the end. I will simultaneously introduce the use of dependent typing to combine abstract syntax and static semantics, since it turns out to be necessary for this encoding technique to work, in contrast to the untyped HOAS ASTs that Twelf supports.

The first step is to introduce what I mean by denotational semantics in Coq:

$$\begin{array}{rcl} \mathsf{tyDenote} & = & \mathsf{fix}\ f\ (t : \mathsf{ty}) : \mathsf{Set} := \\ & & \quad \mathsf{match}\ t\ \mathsf{with} \\ & & \quad\ \ \mid \mathsf{Base} \Rightarrow \mathsf{nat} \\ & & \quad\ \ \mid \mathsf{Arrow}\ t_1\ t_2 \Rightarrow f\ t_1 \rightarrow f\ t_2 \end{array}$$

This is a recursive function definition translating object language types into Coq (meta language) types, as Set is the Coq type of all "program"-level types. Modulo the peculiarities of Coq syntax, $\mathsf{tyDenote}$ corresponds very directly to a standard denotational semantics of a type language. nat is the native Coq type of natural numbers, an arbitrary choice for our domain of base types; and the $\rightarrow$ in the Arrow case is the genuine Coq function type constructor. This definition at first seems both obvious and bizarre; the missing piece is that Coq's higher-order logic really is designed both to be sound and to admit this kind of definition.

With the denotation of types in hand, we're ready to define our higher-order term type:

$$\begin{array}{l} \mathsf{Inductive\ term} : \mathsf{ty} \rightarrow \mathsf{Set} := \\ \quad \mid \mathsf{EVar} : \forall t : \mathsf{ty}.\ \mathsf{tyDenote}\ t \rightarrow \mathsf{term}\ t \\ \quad \mid \mathsf{Lam} : \forall dom, ran : \mathsf{ty}.\ (\mathsf{tyDenote}\ dom \rightarrow \mathsf{term}\ ran) \rightarrow \mathsf{term}\ (\mathsf{Arrow}\ dom\ ran) \\ \quad \mid \mathsf{App} : \forall dom, ran : \mathsf{ty}.\ \mathsf{term}\ (\mathsf{Arrow}\ dom\ ran) \rightarrow \mathsf{term}\ dom \rightarrow \mathsf{term}\ ran \end{array}$$

With this definition, our running example is written as $\mathsf{Lam\ Base\ Base}$ (λx : $\mathsf{tyDenote\ Base}$. $\mathsf{EVar\ Base}\ x$). Coq's type-checker identifies terms up to computational reduction, so the use of $\mathsf{tyDenote\ Base}$ is equivalent to a use of nat; the function really is over natural numbers in general.

The first thing to notice is that after the colon on the first line we now have $\mathsf{ty} \to \mathsf{Set}$ instead of just Set. This indicates that term is an indexed type family of one argument, where the argument tells us the object language type of a term. The usual typing rules for simply-typed lambda calculus have been merged into the description of abstract syntax, as shown in the types of term's constructors, where $\forall$ quantifications over types are used to bind types used as the indices of the argument and result term types.

This definition isn't *quite* traditional HOAS, but it shares the main benefits of that technique. Something very odd has happened in the definition: through the EVar constructor, we allow the injection into our term language of *any arbitrary Coq term* that happens to have a Coq type that is the denotation of some object language type! Naturally, this means that this definition no longer captures the informal notion of simply-typed lambda calculus. Though I won't go into detail about it here, we can remove this deficiency by introducing in Coq a term well-formedness judgment parameterized by a context listing which Coq terms are acceptable arguments to EVar.

The main idea of this representation is that, like in normal HOAS, we represent object language variables with meta language variables. The key change is that these *variables stand for the denotations of terms rather than their syntactic representations.*

While Coq won't allow a meta language type to appear in a function argument position within its own inductive definition, it has no problem with allowing in such a position a previously-defined type that will suffice for describing the *meaning* of a value of the new type. The lambda constructor binds just such a meaning in place of an AST, and such meanings are *used* as arguments to EVar constructors.

Now we are ready to say what terms *mean*, via another denotation function:

$$\begin{array}{rcl} \mathsf{termDenote} & = & \mathsf{fix}\ f\ (t : \mathsf{ty})\ (e : \mathsf{term}\ t) : \mathsf{tyDenote}\ t := \\ & & \quad \mathsf{match}\ e\ \mathsf{with} \\ & & \quad\ \mid \mathsf{EVar}\ \_\ x \Rightarrow x \\ & & \quad\ \mid \mathsf{Lam}\ \_\ \_\ e' \Rightarrow \lambda x.\ f\ (e'\ x) \\ & & \quad\ \mid \mathsf{App}\ \_\ \_\ e_1\ e_2 \Rightarrow (f\ e_1)\ (f\ e_2) \end{array}$$

We see again the fix form for defining recursive functions. This function termDenote takes two arguments, a type t and an expression e of that type. It returns the *meaning* of e, whose type is determined by t via the tyDenote function we defined earlier.

Like for the definition of tyDenote, the body of termDenote's definition probably seems simple enough but also quite surprising. Since we have defined variables to bind meanings to begin with, the denotation of a variable is simply the value that it carries. The denotation of an abstraction is a function that returns the denotation of the body function applied to the meta language function argument. The denotation of an application is the application of the denotations of the two subterms.

This definition of a dynamic semantics for terms has a surprising and very welcome property: large classes of correctness proofs about terms become provable via *atomic proof steps* in Coq's logic. Here's an example to illustrate, in concrete Coq syntax.

```
Definition ty1 : ty := Arrow Base Base.
Definition id : term (Arrow ty1 ty1) :=
  Lam _ _ (fun x => EVar _ x).

Theorem id_correct : forall (e : term ty1),
  termDenote _ (App _ _ id e)
  = termDenote _ e.
Proof.
  trivial.
Qed.
```

First, we define a synonym `ty1` for the type of functions from the base type to itself. Next, we define the identity function `id` over `ty1`. Finally, we want to check that our identity function behaves as we expect, so we prove a theorem that the denotation of `id` applied to any term `e` of the proper type is the same as the denotation of `e` itself.

The single Coq tactic invocation to `trivial` suffices to prove the goal. A single Coq tactic may do a large amount of work and return a very complicated proof in the underlying logic, but in this case, `trivial` finds a very trivial proof indeed. The resulting Curry-Howard-style proof term is simply a function binding the variable `e` whose body is a use of *the reflexivity rule for equality*! As mentioned previously, the Coq type checker effectively evaluates terms using computational rules like beta reduction before commencing with "normal" type checking. Since object language functions have been compiled into meta language functions, and since Coq knows about the computational semantics of its functions, the two sides of the equality do indeed evaluate to the same expression, justifying the use of reflexivity.

This phenomenon is not limited to tiny examples like this. This sort of *proof by reflection* scales up to correctness theorems about object language programs of arbitrary size, allowing proof terms with size linear in the size of the program being verified. Another way of describing the trick that we've pulled here is that the developers of the Coq proof assistant have invested a lot of effort in making it very easy to reason about Coq's logic. What we've done here is *reduced* the questions that we're interested in into questions about that very logic. Even better, the reduction process is *reified in that same logic*, so we can prove "meta-theorems" about it.

Compare these results with how operational semantics has been implemented in formal settings. There, there may be an effective logic program interpreter that can prove these correctness theorems automatically, but it almost certainly outputs proof terms of arbitrary size as the sizes of programs to verify grows; or it doesn't output proof terms at all, and a skeptical checker of a supposed correctness proof needs to run a logic programming engine himself, worrying about the issues of termination and coverage that are at the heart of Twelf's meta-theoretical functionality. Furthermore, when we want to write "meta-proofs" about the dynamic semantics rather than check individual instances of it, the situation becomes even more tangled. Having term evaluation as an atomic operation in our logic goes a long way towards simplifying this kind of proof.

On the other hand, when we want to formalize *syntactic* operations over our HOAS terms, we run into the traditional problems. Even something as simple as a recursive function that computes the size of a term is tricky. Proposed solutions based on modal logic involve changes to our meta language. While something like that may be possible for Coq, I've opted instead to switch to a strategy based on nameless first-order representation, which is the subject of the next subsection. However, I'll keep the main idea from this section: a synergistic combination of dependently-typed ASTs and denotational semantics.

6.1.4 De Bruijn with Dependent Types

This final solution combines the de Bruijn encoding with dependent typing. I've implemented a library called Lambda Tamer for supporting this kind of formalization, and I'll drawn on library code from Lambda Tamer in the rest of Part II. Here is a Lambda Tamer version of term:

$$
\begin{array}{l}
\mathsf{Inductive\ term} : \mathsf{list\ ty} \to \mathsf{ty} \to \mathsf{Set} := \\
\quad \mid \mathsf{EVar} : \forall \Gamma : \mathsf{list\ ty}.\ \forall t : \mathsf{ty}.\ \mathsf{var}\ \Gamma\ t \to \mathsf{term}\ \Gamma\ t \\
\quad \mid \mathsf{Lam} : \forall \Gamma : \mathsf{list\ ty}.\ \forall dom, ran : \mathsf{ty}.\ \mathsf{term}\ (dom :: \Gamma)\ ran \to \mathsf{term}\ \Gamma\ (\mathsf{Arrow}\ dom\ ran) \\
\quad \mid \mathsf{App} : \forall \Gamma : \mathsf{list}\ ty.\ \forall dom, ran : \mathsf{ty}.\ \mathsf{term}\ \Gamma\ (\mathsf{Arrow}\ dom\ ran) \\
\qquad \to \mathsf{term}\ \Gamma\ dom \to \mathsf{term}\ \Gamma\ ran
\end{array}
$$

The var type family is defined in the Lambda Tamer library as follows:

$$
\begin{array}{l}
\mathsf{Inductive\ var}\ (ty : \mathsf{Set}) : \mathsf{list}\ ty \to ty \to \mathsf{Set} := \\
\quad \mid \mathsf{First} : \forall \Gamma : \mathsf{list}\ ty.\ \forall t : ty.\ \mathsf{var}\ (t :: \Gamma)\ t \\
\quad \mid \mathsf{Next} : \forall \Gamma : \mathsf{list}\ ty.\ \forall t, t' : ty.\ \mathsf{var}\ \Gamma\ t \to \mathsf{var}\ (t' :: \Gamma)\ t
\end{array}
$$

$\mathsf{var}\ \Gamma\ t$ can be thought of as a constructive proof that a variable of type t is found in context Γ. However, a var isn't quite a "proof," because it is placed in Set, not Prop. This is important, since it allows variable-manipulating operations to survive extraction. Alternatively, a var is like a natural number with extra typing added to track which element of a list it selects.

We now arrive at the final form for the running example term:

$$\mathsf{Lam\ nil\ Base\ Base\ (EVar\ nil\ Base\ (First\ nil\ Base))}$$

Since we no longer use the meta language typing context to encode the object language typing context, the type family term takes a new first parameter, a list of types representing the context. Since this is a nameless encoding, there is no need to associate names with elements of the context; only order matters. The types of term's constructors quantify over a context Γ, and this context is used in a standard way in determining the argument and result types.

Now we can define a term denotation function similar to the definition from the last sub-section.

$$
\begin{array}{lcl}
\mathsf{termDenote} & = & \mathsf{fix}\ f\ (\Gamma : \mathsf{list\ ty})\ (t : \mathsf{ty})\ (e : \mathsf{term}\ \Gamma\ t) : \mathsf{subst\ tyDenote}\ \Gamma \to \mathsf{tyDenote}\ t := \\
 & & \quad \mathsf{match}\ e\ \mathsf{with} \\
 & & \quad\quad \mid \mathsf{EVar}\ \_\ \_\ x \Rightarrow \lambda\sigma.\ \mathsf{varDenote}\ x\ \sigma \\
 & & \quad\quad \mid \mathsf{Lam}\ \_\ \_\ \_\ e' \Rightarrow \lambda\sigma.\ \lambda x.\ f\ e'\ (\mathsf{SCons}\ x\ \sigma) \\
 & & \quad\quad \mid \mathsf{App}\ \_\ \_\ \_\ e_1\ e_2 \Rightarrow \lambda\sigma.\ (f\ e_1\ \sigma)\ (f\ e_2\ \sigma)
\end{array}
$$

The main change is that termDenote now returns more than just the meaning of the input expression e. Rather, it returns a function from *a substitution giving values to e's free variables* to that meaning. The type family subst of substitutions is taken from the Lambda tamer library. subst is parameterized on the denotation function to use to interpret contexts and on the actual context of interest.

The function varDenote used in the EVar case is also from the same library. It implements the obvious variable lookup operation in the current substitution. The Lam and App cases are similar to before, with the exception that, for Lam, the actual function argument is specified by building a new substitution with the SCons constructor rather than by calling a function returning a term.

We can redo the correctness theorem example from the last sub-section:

```
Definition id : term nil (Arrow ty1 ty1) :=
  Lam _ _ _ (EVar _ _ (First _ _)).

Theorem id_correct : forall (e : term nil ty1),
  termDenote _ _ (App _ _ _ id e) SNil
  = termDenote _ _ e SNil.
```

```
Proof.
  trivial.
Qed.
```

The statement of `id_correct` is almost the same, except that we have to provide the trivial empty substitution `SNil` as an extra argument on each side of the equality. The proof goes through again by reflexivity, with computational reduction taking care of evaluating all of the relevant operations on de Bruijn terms.

6.2 Compilation

The dependent de Bruijn representation, in combination with the library used in a few places in the last section, achieves a nice balance of the benefits of the different approaches surveyed here. The library takes care of many of the tedious operations associated with de Bruijn representation, while we retain the benefits of denotational semantics for trivial Coq proofs of facts about dynamic semantics. In addition, we're now able to write syntactic functions quite easily. For instance, here's a function to compute the tree size of a term:

$$\begin{array}{rcl}
\mathsf{size} & = & \mathsf{fix}\ f\ (\Gamma : \mathsf{list\ ty})\ (t : \mathsf{ty})\ (e : \mathsf{term}\ \Gamma\ t) : \mathsf{nat} := \\
& & \quad \mathsf{match}\ e\ \mathsf{with} \\
& & \quad\ \ \mid \mathsf{EVar}\ _\ _\ _ \Rightarrow 1 \\
& & \quad\ \ \mid \mathsf{Lam}\ _\ _\ _\ e' \Rightarrow 1 + f\ e' \\
& & \quad\ \ \mid \mathsf{App}\ _\ _\ _\ e_1\ e_2 \Rightarrow 1 + f\ e_1 + f\ e_2
\end{array}$$

If this were the only sort of function that we could write that was tricky with HOAS, then there wouldn't be much of an argument for this representation. However, *transformations* between languages are a critical element of programming language theory.

Our new representation makes it very convenient both to code purely syntactic transformations like compilers and to prove their correctness.

I'll present the basic ideas through an example. Let's say our source language contains a "let" binding construct, in addition to the features we've been working with so far. This Coq type definition captures the abstract syntax of the expanded language.

$$
\begin{array}{l}
\textsf{Inductive lterm : list ty} \to \textsf{ty} \to \textsf{Set} := \\
\quad \mid \textsf{LVar} : \forall \Gamma : \textsf{list ty}.\ \forall t : \textsf{ty}.\ \textsf{var}\ \Gamma\ t \to \textsf{lterm}\ \Gamma\ t \\
\quad \mid \textsf{LLam} : \forall \Gamma : \textsf{list ty}.\ \forall dom, ran : \textsf{ty}.\ \textsf{lterm}(dom :: \Gamma)\ ran \to \textsf{lterm}\ \Gamma\ (\textsf{Arrow}\ dom\ ran) \\
\quad \mid \textsf{LApp} : \forall \Gamma : \textsf{list ty}.\ \forall dom, ran : \textsf{ty}. \\
\quad\quad \textsf{lterm}\ \Gamma\ (\textsf{Arrow}\ dom\ ran) \to \textsf{lterm}\ \Gamma\ dom \to \textsf{lterm}\ \Gamma\ ran \\
\quad \mid \textsf{LLet} : \forall \Gamma : \textsf{list ty}.\ \forall bound, result : \textsf{ty}. \\
\quad\quad \textsf{lterm}\ \Gamma\ bound \to \textsf{lterm}\ (bound :: \Gamma)\ result \to \textsf{lterm}\ \Gamma\ result
\end{array}
$$

We can give our "let" terms a denotational semantics in the same style.

$$
\begin{array}{lcl}
\textsf{ltermDenote} & = & \textsf{fix}\ f\ (\Gamma : \textsf{list ty})\ (t : \textsf{ty})\ (e : \textsf{lterm}\ \Gamma\ t) : \textsf{subst tyDenote}\ \Gamma \to \textsf{tyDenote}\ t := \\
& & \quad \textsf{match}\ e\ \textsf{with} \\
& & \quad\quad \mid \textsf{LVar}\ \_\ \_\ x \Rightarrow \lambda\sigma.\ \textsf{varDenote}\ x\ s \\
& & \quad\quad \mid \textsf{LLam}\ \_\ \_\ \_\ e' \Rightarrow \lambda\sigma.\ \lambda x.\ f\ e'\ (\textsf{SCons}\ x\ \sigma) \\
& & \quad\quad \mid \textsf{LApp}\ \_\ \_\ \_\ e_1\ e_2 \Rightarrow \lambda\sigma.\ (f\ e_1\ \sigma)\ (f\ e_2\ \sigma) \\
& & \quad\quad \mid \textsf{LLet}\ \_\ \_\ \_\ e_1\ e_2 \Rightarrow \lambda\sigma.\ f\ e_2\ (\textsf{SCons}\ (f\ e_1\ \sigma)\ \sigma)
\end{array}
$$

The case for LLet evaluates the "let" body in a substitution extended with the denotation of the bound variable's definition.

We know that "let" is actually superfluous in simply-typed lambda calculus, since any "let" expression can be represented by a lambda abstraction immediately applied to the bound variable definition from the "let." Looking to simplify terms, we decide to compile these expressions away through a translation to the original calculus. This function implements that compilation.

$$
\begin{array}{rcl}
\mathsf{compile} & = & \mathsf{fix}\ f\ (\Gamma : \mathsf{list\ ty})\ (t : \mathsf{ty})\ (e : \mathsf{lterm}\ \Gamma\ t) : \mathsf{term}\ \Gamma\ t := \\
& & \quad \mathsf{match}\ e\ \mathsf{with} \\
& & \quad\quad \mid \mathsf{LVar}\ \_\ \_\ x \Rightarrow \mathsf{EVar}\ x \\
& & \quad\quad \mid \mathsf{LLam}\ \_\ \_\ \_\ e' \Rightarrow \mathsf{Lam}\ (f\ e') \\
& & \quad\quad \mid \mathsf{LApp}\ \_\ \_\ \_\ e_1\ e_2 \Rightarrow \mathsf{App}\ (f\ e_1)\ (f\ e_2) \\
& & \quad\quad \mid \mathsf{LLet}\ \_\ \_\ \_\ e_1\ e_2 \Rightarrow \mathsf{App}\ (\mathsf{Lam}\ (f\ e_2))\ (f\ e_1)
\end{array}
$$

Once the obstacle of learning Coq's concrete syntax is overcome, it's easy to see that this code is just the obvious compilation function. The fact that this definition makes it past the Coq type-checker serves as a proof that it is *type-preserving*. We would also like to check that it is *semantics-preserving*; that is, that every output of `compile` has the same denotation as the corresponding input. The following Coq code states and proves this theorem.

```
Theorem compile_correct : forall (G : list ty) (t : ty) (e : lterm G t)
  (s : subst tyDenote G),
  termDenote _ _ (compile _ _ e) s
  = ltermDenote _ _ e s.
Proof.
  induction e; simplify.
Qed.
```

The theorem is generalized to deal with open terms by quantifying over legal substitutions for the free variables of the input term `e`. The actual proof of the theorem is not as simple as before, since Coq computational reduction can't replace an inductive argument, but we don't need to provide that much more input to the proof assistant. We give the top-level structure of the argument to use with the `induction` tactic and call a tactic called `simplify` defined in the Lambda Tamer library. This tactic is designed for automatic proof of goals that show up in proofs of compiler correctness. Its basic strategy is to perform computational reduction everywhere and try all bounded-length sequences of rewritings in the goal based on universally-quantified equalities among the hypotheses. In this case, `simplify` discharges all of the subgoals of the induction. Crucially, the only interesting case of the compilation, the "let" case, was proved completely automatically, since computational reduction makes it obvious. Even better, the Curry-Howard-style proof term generated by this interaction is only about 3000 characters long when printed, including extra type annotations needed for decidable proof checking.

6.3 Related Work

Crary proposed the method of type theoretical semantics [24], giving a non-computer-formalized embedding of an expressive typed lambda calculus into the type theory of the Nuprl system. Lee et al. [55] mechanize the semantics of Standard ML operationally in Twelf. The POPLmark Challenge [4] is a benchmark initiative for language formalization that has prompted many interesting solutions to its first challenge problem. None of these projects involve formalizations of compiler correctness proofs.

Recent work has implemented tagless interpreters [71] using generalized algebraic datatypes and other features related to dependent typing. Tagless interpreters have much in common with my approach to denotational semantics in Coq, but I am not aware of any proofs beyond type safety carried out in such settings.

The relationship between my proposed approach and traditional styles based on operational semantics is quite analogous to that between the "semantic" approach to foundational proof-carrying code [2] and the "syntactic" approach [40].

Chapter 7

A Certified Type-Preserving Compiler

7.1 Introduction

Compilers are some of the most complicated pieces of widely-used software. This fact has unfortunate consequences, since almost all of the guarantees we would like our programs to satisfy depend on the proper workings of our compilers. Therefore, proofs of compiler correctness are at the forefront of useful formal methods research, as results here stand to benefit many users. Thus, it is not surprising that recently and historically there has been much interest in this class of problems. This chapter is a report on my foray into that area and the new techniques that I developed, based on the foundation laid in the previous chapter.

One interesting compiler paradigm is *type-directed compilation*, embodied in, for instance, the TIL Standard ML compiler [85]. Where traditional compilers use relatively type-impoverished intermediate languages, TIL employed *typed intermediate languages* such that every intermediate program had a typing derivation witnessing its safety, up until the last few phases of compilation. This type information can be used to drive important optimizations, including *nearly tag-free garbage collection*, where the final binary comes with a table giving the type of each register or stack slot at each program point where the garbage collector may be called. As a result, there is no need to use any dynamic typing scheme for values in registers, such as tag bits or boxing.

Most of the intricacies of TIL stemmed from runtime passing of type information to support polymorphism. In the work I present here, I instead picked the modest starting point of simply-typed lambda calculus, but with the same goal: to compile the programs of this calculus into an idealized assembly language that uses nearly tag-free garbage collection. I achieve this result by a series of six type-directed translations, with their typed target languages maintaining coarser and coarser grained types as we proceed. Most importantly, I prove *semantics preservation* for each translation and compose these results into a machine-checked correctness proof for the compiler. To my knowledge, there exists no other computer formalization of such a proof for a type-preserving compiler.

At this point, the reader may be dubious about just how involved a study of simply-typed lambda calculus can be. I hope that the exposition in the remainder of this chapter will justify the interest of the domain.

$$
\begin{array}{rrll}
\text{Types} & \tau & ::= & \mathsf{N} \mid \tau \to \tau \\
\\
\text{Natural numbers} & n & & \\
\text{Variables} & x, y, z & & \\
\text{Terms} & e & ::= & n \mid x \mid e\ e \mid \lambda x : \tau.\ e
\end{array}
$$

Figure 7.1: Source language syntax

$$
\begin{array}{rrll}
\text{Registers} & r & & \\
\text{Operands} & o & ::= & r \mid n \mid \mathsf{new}\ \langle \vec{r}, \vec{r} \rangle \mid \mathsf{read}\ \langle r, n \rangle \\
\text{Instructions} & i & ::= & r := o; i \mid \mathsf{jump}\ r \\
\text{Programs} & p & ::= & \langle \vec{i}, i \rangle
\end{array}
$$

Figure 7.2: Target language syntax

7.1.1 Task Description

The source language of the compiler is the familiar simply-typed lambda calculus, whose syntax is shown in Figure 7.1. Application, the third production for e, associates to the left, as usual. I will elaborate shortly on the language's semantics.

The target language is an idealized assembly language, with infinitely many registers and memory cells, each of which holds unbounded natural numbers. I model interaction with a runtime system through special instructions. The syntax of the target language is given in Figure 7.2.

As instruction operands, we have registers, constant naturals, and allocation and reading of heap-allocated records. The new operand takes two arguments: a list of root registers and a list of registers holding the field values for the object to allocate. The semantics of the target language are such that new is allowed to perform garbage collections at will, and the root list is the usual parameter required for sound garbage collection. In fact, we will let new rearrange memory however it pleases, as long as the result is indistinguishable from having simply allocated a new record, from the point of view of the specified roots. The read instruction determines which runtime system-specific procedure to use to read a given field of a record.

A program is a list of basic blocks plus an initial basic block, where execution begins. A basic block is a sequence of register assignments followed by an indirect jump. The basic blocks are indexed in order by the natural numbers for the purposes of these jumps. We will additionally consider that a jump to the value 0 indicates that the program should halt, and we distinguish one register that is said to hold the program's result at such points.

We are now ready to state informally the theorem whose proof is the goal of this work:

Theorem 1 (Informal statement of compiler correctness) *Given a term e of the simply-typed lambda calculus of type N, the compilation of e is an assembly program that, when run, terminates with the same result as we obtain by running e.*

The compiler itself is implemented entirely within Coq, as in the certified verifiers project from Part I. Through program extraction, we obtain a traditional executable version of the compiler. I ignore issues of parsing in this work, so the compiler must be composed with a parser. Assuming a fictitious machine that runs our idealized assembly language, the only remaining piece to be added is a pretty-printer from the abstract syntax tree representation of assembly programs to whatever format that machine requires.

7.1.2 Contributions

I summarize the contributions of this work as follows:

- It includes what is to my knowledge the first total correctness proof of an entire type-preserving compiler.
- It gives the proof using denotational semantics, in contrast to the typical use of operational semantics in related work.
- The whole formal development is carried out in a completely rigorous way with the Coq proof assistant [8], yielding a proof that can be checked by a machine using a relatively small trusted code base. My approach is based on the general methodology for representing variable binding and denotation functions in Coq outlined in Chapter 6.
- I sketch a generic programming system AutoSyntax that I have developed for automating construction of syntactic helper functions over de Bruijn terms [28], as well as generic correctness proofs about these functions. In addition, I present the catalogue of generic functions that I found sufficient for this work. This chapter contains a summary of AutoSyntax, while Chapter 8 is devoted to its implementation.
- Finally, I add to the list of pleasant consequences of making your compiler type-directed or type-preserving. In particular, I show how dependently-typed formalizations of type-preserving compilers admit particularly effective automated proof methods, driven by type information.

7.1.3 Outline

In the next section, I present the compiler's intermediate languages and elaborate on the semantics of the source and target languages. Following that, I run through the basic elements of implementing a compiler pass, noting challenges that arise in the process. Each of the next sections addresses one of these challenges and my solution to it. I give brief examples of applying Chapter 6's methodology to the languages and transformations relevant here, and I discuss how to use generic programming to simplify programming with dependently-typed abstract syntax trees and how to apply automated theorem proving to broad classes of proof obligations arising in certification of type-preserving compilers. After this broad discussion, I spend some time discussing interesting features of particular parts of the formalization. Finally, I provide some statistics on the implementation, discuss related work, and conclude.

7.2 The Languages

In this section, I present the source, intermediate, and target languages, along with their static and dynamic semantics. The language progression is reminiscent of that from the paper "From System F to Typed Assembly Language" [66]. The main differences stem from the fact that I am interested in meaning preservation, not just type safety. This distinction makes the task both harder and easier. Naturally, semantics preservation proofs are more difficult than type preservation proofs. However, the fact that we construct such a detailed understanding of program behavior allows us to retain less type information in later stages of the compiler.

The mechanics of formalizing these languages in Coq is the subject of later sections. I will stick to a "pencil and paper formalization" level of detail in this section. I try to use standard notations wherever possible. The reader can rest assured that anything that seems ambiguous in the presentation here is clarified in the mechanization.

7.2.1 Source

The syntax of our source language (called "Source" hereafter) was already given in Section 7.1.1. The type system is the standard one for simply-typed lambda calculus, with judgments of the form $\Gamma \vdash e : \tau$ which mean that, with respect to the type assignment to free variables provided by Γ, term e has type τ.

Following usual conventions, I require that, for any typing judgment that may be stated, let alone verified to hold, all free and bound variables are distinct, and all free variables of the term appear in the context. In the implementation, the first condition is discharged by using de Bruijn indices, and the second condition is covered by the dependent typing rules I will assign to terms.

Next we need to give Source a dynamic semantics. For this and all the other languages, I opted to use denotational semantics. The utility of this choice for compiler correctness proofs has been understood for a while, as reifying a program phrase's meaning as a mathematical object makes it easy to transplant that meaning to new contexts in modeling code transformations.

The semantics for Source is shown in Figure 7.3. The type of the term denotation function (used in expressions of the form $[\![e]\!]$) indicates that it is a function whose domain is *typing derivations* for terms and whose range depends on the particular Γ and τ that

$$
\begin{array}{rcl}
[\![\tau]\!] & : & \text{types} \rightarrow \text{sets} \\
[\![\mathsf{N}]\!] & = & \mathbb{N} \\
[\![\tau_1 \rightarrow \tau_2]\!] & = & [\![\tau_1]\!] \rightarrow [\![\tau_2]\!] \\
\\
[\![\Gamma]\!] & : & \text{contexts} \rightarrow \text{sets} \\
[\![\cdot]\!] & = & \mathsf{unit} \\
[\![\Gamma, x : \tau]\!] & = & [\![\Gamma]\!] \times [\![\tau]\!] \\
\\
[\![e]\!] & : & [\Gamma \vdash e : \tau] \rightarrow [\![\Gamma]\!] \rightarrow [\![\tau]\!] \\
[\![n]\!]\sigma & = & \overline{n} \\
[\![x]\!]\sigma & = & \sigma(x) \\
[\![e_1\ e_2]\!]\sigma & = & [\![e_1]\!]\sigma\ [\![e_2]\!]\sigma \\
[\![\lambda x : \tau.\ e]\!]\sigma & = & \lambda x : [\![\tau]\!].\ [\![e]\!](\sigma, x)
\end{array}
$$

Figure 7.3: Source language dynamic semantics

appear in that derivation. As a result, we define denotations only for well-typed terms. Every syntactic class is being compiled into a single meta language, Coq's Calculus of Inductive Constructions (CIC) [8].

We first give each Source type a meaning by a recursive translation into sets. Note that, throughout this chapter, I overload constructs like the function type constructor $\rightarrow$ that are found in both the object languages and the meta language CIC, in an effort to save the reader from a deluge of different arrows. The variety of arrow in question should always be clear from context. With this convention in mind, the type translation for Source is entirely unsurprising. $\mathbb{N}$ denotes the mathematical set of natural numbers, as usual.

We give a particular type theoretical interpretation of typing contexts as tuples, and we then interpret a term that has type τ in context Γ as a function from the denotation of Γ into the denotation of τ. The translation here is again entirely standard. For the sake

$$
\begin{array}{rrcl}
\text{Operands} & o & ::= & n \mid x \mid \lambda x : \tau .\ e \\
\text{Terms} & e & ::= & \mathsf{let}\ x = o\ \mathsf{in}\ e \mid \mathsf{throw}\ \langle o \rangle \mid x\ y\ z
\end{array}
$$

Figure 7.4: Syntax of the Linear language

of conciseness, I allow myself to be a little sloppy with notations like $\sigma(x)$, which denotes the proper projection from the tuple σ, corresponding to the position x occupies in Γ. I note that I use the *meta language's lambda binder* to encode the object language's lambda binder in a natural way, in an example closely related to higher-order abstract syntax [75].

7.2.2 Linear

Our first compilation step is to convert Source programs into a form that makes *execution order* explicit. This kind of translation is associated with continuation-passing style (CPS), and the composition of the first two translations accomplishes a transformation to CPS. The result of the first translation is the Linear language, and I now give its syntax. It inherits the type language of Source, though we interpret the types differently.

Figure 7.4 presents the syntax of Linear. The terms are linearized in the sense that they are sequences of binders of primitive operands to variables, followed by either a "throw to the current continuation" or a function call. The function call form shows that functions take two arguments: first, the normal argument; and second, a function to call with the result. The function and both its arguments must be variables, perhaps bound with earlier lets.

As Linear has an entirely unsurprising static semantics, I omit its details and proceed to the denotational semantics. Recall, however, that the denotation functions are only defined over well-typed terms.

$$
\begin{array}{rcl}
[\![\tau]\!] & : & \text{types} \to \text{sets} \\
[\![\mathsf{N}]\!] & = & \mathbb{N} \\
[\![\tau_1 \to \tau_2]\!] & = & [\![\tau_1]\!] \to ([\![\tau_2]\!] \to \mathbb{N}) \to \mathbb{N} \\
\\
[\![o]\!] & : & [\Gamma \vdash o : \tau] \to [\![\Gamma]\!] \to [\![\tau]\!] \\
[\![n]\!]\sigma & = & \overline{n} \\
[\![x]\!]\sigma & = & \sigma(x) \\
[\![\lambda x : \tau.\ e]\!]\sigma & = & \lambda x : [\![\tau]\!].\ [\![e]\!](\sigma, x) \\
\\
[\![e]\!] & : & [\Gamma \vdash o : \tau] \to [\![\Gamma]\!] \to ([\![\tau]\!] \to \mathbb{N}) \to \mathbb{N} \\
[\![\mathsf{let}\ x = o\ \mathsf{in}\ e]\!]\sigma & = & [\![e]\!](\sigma, [\![o]\!]\sigma) \\
[\![\mathsf{throw}\ \langle o \rangle]\!]\sigma & = & \lambda k.\ k([\![o]\!]\sigma) \\
[\![x\ y\ z]\!]\sigma & = & \lambda k.\ \sigma(x)\ (\sigma(y))\ (\lambda v.\ k(\sigma(z)(v)))
\end{array}
$$

Figure 7.5: Dynamic semantics of the Linear language

Figure 7.5 shows the dynamic semantics. We choose the natural numbers as the result type of programs. Functions are interpreted as accepting continuations that return naturals when given a value of the range type. The let case relies on the implicit fact that the newly-bound variable x falls at the end of the proper typing context for the body e. The most interesting case is the last one listed, that for function calls. We call the provided function with a new continuation created by composing the current continuation with the continuation supplied through the variable z.

I use Linear as my first intermediate language instead of going directly to standard CPS because the presence of distinguished throw terms makes it easier to optimize term representation by splicing terms together. This separation is related to the issue of "administrative redexes" in standard two-phase CPS transforms [79].

$$\begin{array}{rcll} \text{Types} & \tau & ::= & \mathsf{N} \mid \vec{\tau} \rightarrow \mathbb{N} \\ \text{Operands} & o & ::= & n \mid x \mid \lambda \vec{x} : \vec{\tau}.\ e \\ \text{Terms} & e & ::= & \mathsf{let}\ x = o\ \mathsf{in}\ e \mid x\ \vec{y} \end{array}$$

Figure 7.6: Syntax of the CPS language

$$\begin{array}{rcll} \text{Types} & \tau & ::= & \mathsf{N} \mid \vec{\tau} \rightarrow \mathbb{N} \mid \bigotimes \vec{\tau} \mid \vec{\tau} \times \vec{\tau} \rightarrow \mathbb{N} \\ \text{Operands} & o & ::= & n \mid x \mid \langle x, \vec{y} \rangle \mid \pi_i x \\ \text{Terms} & e & ::= & \mathsf{let}\ x = o\ \mathsf{in}\ e \mid x\ \vec{y} \\ \text{Programs} & p & ::= & \mathsf{let}\ x = (\lambda \vec{y} : \vec{\tau}.\ e)\ \mathsf{in}\ p \mid e \end{array}$$

Figure 7.7: Syntax of the CC language

7.2.3 CPS

The next transformation finishes the job of translating Linear into genuine CPS form, and I call this next target language CPS.

Figure 7.6 shows its syntax. Compared to Linear, the main differences we see are that functions may now take multiple arguments and that we have collapsed throw and function call into a single construct. In my intended use of CPS, functions will either correspond to source-level functions and take two arguments, or they will correspond to continuations and take single arguments. The type language has changed to reflect that functions no longer return, but rather they lead to final natural number results.

I omit discussion of the semantics of CPS, since the changes from Linear are quite incremental.

7.2.4 CC

The next thing the compiler does is closure convert CPS programs, hoisting all function definitions to the top level and changing those functions to take records of their

$$
\begin{array}{rcll}
\text{Types} & \tau & ::= & \mathsf{N} \mid \mathsf{ref} \mid \vec{\tau} \to \mathbb{N} \\
\text{Operands} & o & ::= & n \mid x \mid \langle n \rangle \mid \mathsf{new}\, \langle \vec{x} \rangle \mid \pi_i x \\
\text{Terms} & e & ::= & \mathsf{let}\ x = o\ \mathsf{in}\ e \mid x\ \vec{y} \\
\text{Programs} & p & ::= & \mathsf{let}\ (\lambda \vec{y} : \vec{\tau}.\ e)\ \mathsf{in}\ p \mid e
\end{array}
$$

Figure 7.8: Syntax of the Alloc language

free variables as additional arguments. I call the result language CC, and Figure 7.7 shows its syntax.

We have two new type constructors: $\bigotimes \vec{\tau}$ is the multiple-argument product type of records whose fields have the types given by $\vec{\tau}$. $\vec{\tau} \times \vec{\tau} \to \mathbb{N}$ is the type of *code pointers*, specifying first the type of the closure environment expected and second the types of the regular arguments.

Among the operands, we no longer have an anonymous function form. Instead, we have $\langle x, \vec{y} \rangle$, which indicates the creation of a closure with code pointer x and environment elements $\vec{y}$. These environment elements are packaged atomically into a record, and the receiving function accesses the record's elements with projection operand form $\pi_i x$. This operand denotes the ith element of record x.

While we have added a number of features, there are no surprises encountered in adapting the previous semantics, so I will proceed to the next language.

7.2.5 Alloc

The next step in compilation is to make allocation of products and closures explicit. Since giving denotational semantics to higher-order imperative programs is tricky, I decided to perform "code flattening" as part of the same transformation. That is, I move to first-order programs with fixed sets of numbered code blocks, and every function call refers to one of these blocks by number. Figure 7.8 shows the syntax of this language, called Alloc.

The first thing to note is that this phase is the first in which we lose type information: we have a single type ref for all heap references, forgetting what we know about record field types. To compensate for this loss of information, we will give Alloc programs a semantics that allows them to fail when they make "incorrect guesses" about the types of heap cells.

Our set of operands now includes both natural number constants n and code pointer constants $\langle n \rangle$. We also have a generic record allocation construct in place of the old closure constructor, and we retain projections from records.

This language is the first to take a significant departure from its predecessors so far as dynamic semantics is concerned. The key difference is that Alloc admits non-terminating programs. While variable scoping prevented cycles of function calls in CC and earlier, we lose that restriction with the move to a first-order form. This kind of transformation is inevitable at some point, since our target assembly language has the same property.

Domain theory provides one answer to the questions that arise in modeling non-terminating programs denotationally, but it is difficult to use the classical work on domain theory in the setting of constructive type theory. One very effective alternative comes in the form of the *co-inductive types* [37] that Coq supports. A general knowledge of this class of types is not needed to understand what follows. I will confine my attention to one co-inductive type, a sort of *possibly-infinite streams*. We define this type with the infinite closure of this grammar:

$$\begin{array}{rcl}
\mathbb{C} & = & \{\mathsf{Traced}, \mathsf{Untraced}\} \times \mathbb{N} \\
\mathbb{M} & = & \mathsf{list}\ \ (\mathsf{list}\ \ \mathbb{C}) \\
\mathsf{tagof}(\mathsf{N}) & = & \mathsf{Untraced} \\
\mathsf{tagof}(\mathsf{ref}) & = & \mathsf{Traced} \\
\mathsf{tagof}(\vec{\tau} \to \mathbb{N}) & = & \mathsf{Untraced}
\end{array}$$

Figure 7.9: Domains for describing tagged heaps

$$\text{Traces} \quad T \quad ::= \quad n \mid \perp \mid \star, T$$

In other words, a *trace* is either an infinite sequence of stars or a finite sequence of stars followed by a natural number or a bottom value. The first of these possibilities will be the denotation of a non-terminating program, the second will denote a program returning an answer, and the third will denote a program that "crashes."

Now we are ready to start modeling the semantics of Alloc. First, we need to fix a representation of the heap. I chose an abstract representation that identifies heaps with lists of lists of tagged fields, standing for the set of finite-size records currently allocated. Each field consists of a tag, telling whether or not it is a pointer; and a data component. As we move to lower languages, these abstract heaps will be mapped into more conventional flat, untyped heaps. In Figure 7.9, I define some domains to represent heaps, along with a function for determining the proper tag for a type.

Next we give a semantics to operands. I make two different choices here than I did before. First, I allow execution of operands to *fail* when a projection reads a value from the heap and finds that it has the wrong tag. Second, the denotations of operands must be

$$
\begin{aligned}
[\![o]\!] &: [\Gamma \vdash o : \tau] \to [\![\Gamma]\!] \to \mathbb{M} \to (\mathbb{M} \times \mathbb{N}) \cup \{\bot\} \\
[\![n]\!]\sigma m &= (m, \overline{n}) \\
[\![x]\!]\sigma m &= (m, \sigma(x)) \\
[\![\langle n \rangle]\!]\sigma m &= (m, \overline{n+1}) \\
[\![\mathsf{new}\ \langle \vec{x} \rangle]\!]\sigma m &= (m \oplus [\sigma(\vec{x})], |m|) \\
[\![\pi_i x]\!]\sigma m &= \textbf{if } m_{\sigma(x),i} = (\mathsf{tagof}(\tau), v), \textbf{ then}\text{: } (m, v) \\
&\ \ \textbf{else}\text{: } \bot
\end{aligned}
$$

Figure 7.10: Dynamic semantics of operands for the Alloc language

$$
\begin{aligned}
[\![e]\!] &: [\Gamma \vdash e : \tau] \to [\![\Gamma]\!] \to \mathbb{M} \\
&\ \ \to (\mathbb{M} \times \mathbb{N} \times \mathsf{list}\ \ \mathbb{N}) \cup \{\bot\} \\
[\![\mathsf{let}\ x = o\ \mathsf{in}\ e]\!]\sigma m &= \textbf{if } [\![o]\!]\sigma m = (m', v) \textbf{ then}\text{: } [\![e]\!](\sigma, v)m' \\
&\ \ \textbf{else}\text{: } \bot \\
[\![x\ \vec{y}]\!]\sigma m &= (m, \sigma(x), \sigma(\vec{y}))
\end{aligned}
$$

Figure 7.11: Dynamic semantics of terms for the Alloc language

heap transformers, taking the heap as an extra argument and returning a new one to reflect any changes. I also set the convention that program counter 0 denotes the distinguished top-level continuation of the program that, when called, halts the program with its first argument as the result. Any other program counter $n + 1$ denotes the nth function defined in the program.

Figure 7.10 gives the dynamic semantics for primitive operands. I write $\oplus$ to indicate list concatenation, and the notation $\sigma(\vec{x})$ to denote looking up in σ the value of each variable in $\vec{x}$, forming a list of results where each is tagged appropriately based on the type of the source variable. The new case returns the length of the heap because I represent heap record addresses with their zero-based positions in the heap list.

$$
\begin{array}{rcl}
[\![p]\!] & : & [\Gamma \vdash \mathsf{let}\ \vec{x}\ \mathsf{in}\ e] \to [\![\Gamma]\!] \to \mathbb{M} \to \mathsf{Trace} \\
[\![\mathsf{let}\ \vec{x}\ \mathsf{in}\ e]\!]\sigma m & = & \textbf{if } [\![e]\!]\sigma m = (m', 0, v :: \vec{n}) \textbf{ then: } v \\
& & \textbf{else if } [\![e]\!]\sigma m = (m', pc + 1, \vec{n}) \textbf{ then:} \\
& & \quad \textbf{if } \vec{x}_{pc} = \lambda \vec{y} : \vec{\tau}.\ e' \textbf{ and } |\vec{y}| = |\vec{n}| \textbf{ then:} \\
& & \quad\quad \star, [\![\mathsf{let}\ \vec{x}\ \mathsf{in}\ e']\!]\vec{n}m' \\
& & \quad \textbf{else: } \bot \\
& & \textbf{else: } \bot
\end{array}
$$

Figure 7.12: Dynamic semantics of programs for the Alloc language

Terms are more complicated. While one might think of terms as potentially non-terminating, I take a different approach here. The denotation of every term is a terminating program that returns *the next function call that should be made*, or signals an error with a $\bot$ value. More precisely, a successful term execution returns a tuple of a new heap, the program counter of the function to call, and a list of natural numbers that are to be the actual arguments. Figure 7.11 gives the details.

Finally, we come to the programs, where we put our trace domain to use, as shown in Figure 7.12. Since we have already converted to CPS, there is no need to consider any aspect of a program's behavior but its result. Therefore, I interpret programs as functions from heaps to traces. I write :: for the binary operator that concatenates a new element to the head of a list, and I write $\vec{x}_{pc}$ for the lookup operation that extracts from the function definition list $\vec{x}$ the pcth element.

Though I have not provided details here, Coq's co-inductive types come with some quite stringent restrictions, designed to prevent unsound interactions with proofs. My definition of program denotation is designed to satisfy those restrictions. The main idea here is that functions defined as *co-fixed points* must be "sufficiently productive"; for our

$$
\begin{array}{rrcl}
\text{Types} & \tau & ::= & \mathsf{N} \mid \mathsf{ref} \mid \Delta \to \mathbb{N} \\
\text{Typings} & \Delta & = & \mathbb{N} \to \tau \\
\text{Registers} & r & & \\
\text{Operands} & o & ::= & n \mid r \mid \langle n \rangle \mid \mathsf{new}\ \langle \vec{r} \rangle \mid \pi_i r \\
\text{Terms} & e & ::= & r := o; e \mid \mathsf{jump}\ r \\
\text{Programs} & p & ::= & \mathsf{let}\ e\ \mathsf{in}\ p \mid e
\end{array}
$$

Figure 7.13: Syntax of the Flat language

trace example, a co-recursive call may only be made after at least one token has already been added to the stream in the current call. This restriction is the reason for including the seemingly information-free stars in my definition of traces. As we have succeeded in drafting a definition that satisfies this requirement, we are rewarded with a function that is not just an arbitrary relational specification, but rather a *program* that Coq is able to execute, resulting in a Haskell-style lazy list.

7.2.6 Flat

We are now almost ready to move to assembly language. Our last stop before then is the Flat language, which differs from assembly only in maintaining the abstract view of the heap as a list of tagged records. We do away with variables and move instead to an infinite bank of registers. Function signatures are expressed as maps from natural numbers to types, and Flat programs are responsible for shifting registers around to compensate for the removal of the built-in stack discipline that variable binding provided.

Figure 7.13 shows the syntax. In the style of Typed Assembly Language, to each static point in a Flat program we associate a typing Δ expressing our expectations of register types whenever that point is reached dynamically. It is crucial that we keep this typing information, since we will use it to create garbage collector root tables in the next and final transformation.

Since there is no need to encode any variable binding for Flat, its denotational semantics is entirely routine. The only exception is that we re-use the trace semantics from Alloc.

7.2.7 Asm

Our idealized assembly language Asm was already introduced in Section 7.1.1. I have by now already provided the main ingredients needed to give it a denotational semantics. We re-use the trace-based approach from the last two languages. The difference we must account for is the shift from abstract to concrete heaps, as well as the fact that Asm is parametric in a runtime system.

I should make it clear that I have not verified any runtime system or garbage collector, but only stated conditions that they ought to satisfy and proved that those conditions imply the correctness of the compiler. Recent work on formal certification of garbage collectors [60] gives hope that the task I have omitted is not insurmountable.

In more detail, my formalization of Asm starts with the definition of the domain $\mathbb{H}$ of concrete heaps:

$$\mathbb{H} \quad = \quad \mathbb{N} \to \mathbb{N}$$

A runtime system provides new and read operations. The read operation is simpler:

$$\mathsf{read} \quad : \quad \mathbb{H} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

For a heap, a pointer to a heap-allocated record, and a constant field offset within that record, read should return that field's current value. Runtime systems may make different decisions on concrete layout of records. For instance, there are several ways of including information on which fields are pointers. Note that read is an arbitrary Coq function, not a sequence of assembly instructions, which should let us reason about many different runtime system design decisions within our framework.

The new operation is more complicated, as it is designed to facilitate garbage collector operation.

$$\begin{aligned} \mathsf{new} \quad : \quad & \mathbb{H} \times \mathsf{list}\ \mathbb{N} \times \mathsf{list}\ \mathbb{N} \\ & \rightarrow \mathbb{H} \times \mathsf{list}\ \mathbb{N} \times \mathbb{N} \end{aligned}$$

Its arguments are the current heap, the values to use to initialize the fields of the record being allocated, and the values of all live registers holding pointer values. The idea is that, in the course of fulfilling the allocation request, the runtime system may rearrange memory however it likes, so long as things afterward look the same as before to any type-safe program, from the perspective of the live registers. A return value of new gives the modified heap, a fresh set of values for the pointer-holding registers (i.e., the garbage collection roots, which may have been moved by a copying collector), and a pointer to the new record in the new heap.

To state the logical conditions imposed on runtime systems, we will need to make a definition based on the abstract heap model of earlier intermediate languages:

Definition 1 (Pointer isomorphism) *We say that pointers $p_1, p_2 \in \mathbb{N}$ are isomorphic with respect to abstract heaps $m_1, m_2 \in \mathbb{M}$ iff:*

1. *The p_1th record of m_1 and p_2th record of m_2 have the same number of fields.*

2. *If the ith field of the p_1th record of m_1 is tagged* Untraced*, then the ith field of the p_2th record of m_2 is also tagged* Untraced*, and the two fields have the same value.*

3. *If the ith field of the p_1th record of m_1 is tagged* Traced*, then the ith field of the p_2th record of m_2 is also tagged* Traced*, and the two fields contain isomorphic pointers.*

The actual definition is slightly more involved but avoids this unqualified self-reference.

To facilitate a parametric translation soundness proof, a candidate runtime system is required to provide a concretization function:

$$\gamma \quad : \quad \mathbb{M} \to \mathbb{H}$$

For an abstract heap m, $\gamma(m)$ is the concrete heap with which the runtime system's representation conventions associate it. I also overload γ to stand for a different function for concretizing pointers. I abuse notation by applying γ to various different kinds of objects, where it's clear how they ought to be converted from abstract to concrete in terms of the two primary γ translations; and, while some of these γ functions really need to take the abstract heap as an additional argument, I omit it for brevity where the proper value is clear from context. The runtime system must come with proofs that its new and read operations satisfy the appropriate commutative diagrams with these concretization functions.

To give a specific example, I show the more complicated condition between the two operations, that for new.

Theorem 2 (Correctness of a new implementation) *For any abstract heap m, tagged record field values $\vec{v}$ (each in $\mathbb{C}$), and register root set values $\vec{rs}$, there exist new abstract* *heap m', new root values $\vec{rs}'$, and new record address a such that:*

1. $\mathsf{new}(\gamma(m), \gamma(\vec{v}), \gamma(\vec{rs})) = (\gamma(m'), \gamma(\vec{rs}'), \gamma(a))$

2. *For every pair of values p and p' in $\vec{rs}$ and $\vec{rs}'$, respectively, p and p' are isomorphic with respect to $m \oplus [\vec{v}]$ and m'.*

3. *$|m|$ and a are isomorphic with respect to $m \oplus [\vec{v}]$ and m'.*

To understand the details of the last two conditions, recall that, in the abstract memory model, we allocate new records at the end of the heap, which is itself a list of records. The length of the heap before an allocation gives the proper address for the next record to be allocated.

7.3 Implementation Strategy Overview

Now that I have sketched the compiler's basic structure, I move to introducing the ideas behind its implementation in the Coq proof assistant. In this section, I single out the first compiler phase and walk through its implementation and proof at a high level, noting the fundamental challenges that we encounter along the way. I summarize my solution to each challenge and then discuss each in more detail in a later section.

Recall that our first phase is analogous to the first phase of a two-phase CPS transform. We want to translate the Source language to the Linear language.

Of course, before we can begin writing the translation, we need to represent our languages! Thus, our first challenge is to choose a representation of each language using Coq types. I used the last chapter to present my approach to this in a more general setting.

The key design decisions were use of *dependently-typed abstract syntax* and *denotational semantics*.

Now we can begin writing the translation from Source to Linear. We have some choices about how to represent translations in Coq, but, with our previous language representation choice, it is quite natural to represent translations as dependently-typed Coq functions. Coq's type system will ensure that, when a translation is fed a well-typed input program, it produces a well-typed output program. We can use Coq's *program extraction* facility to produce OCaml versions of our translations by erasing their dependently-typed parts.

This strategy is very appealing, and it is the one I chose, but it is not without its inconveniences. Since we represent terms as their typing derivations, standard operations like bringing a new, unused variable into scope are not no-ops like they are with some other binding representations. These variable operations turn out to correspond to standard theorems about typing contexts. For instance, bringing an unused variable into scope corresponds to a *weakening* lemma. These syntactic functions are tedious to write for each new language, especially when dealing with strong dependent types. Moreover, their implementations have little to do with details of particular languages. As a result, I have been able to create a generic programming system that produces them automatically for arbitrary languages satisfying certain criteria. It not only produces the functions automatically, but it also produces proofs that they commute with arbitrary compositional denotation functions in the appropriate, function-specific senses.

Now assume that we have our translation implemented. Its type ensures that it

```
Inductive type : Set :=
  | Nat : type
  | Arrow : type → type → type
```

Figure 7.14: Type language for the first two term languages

preserves well-typedness, but we also want to prove that it preserves meaning. What proof technique should we use? The technique of logical relations [78] is the standard for this sort of task. We characterize the relationships between program entities and their compilations using relations defined recursively on type structure. Usual logical relations techniques for denotational semantics translate very effectively into our setting, and we are able to take good advantage of the expressiveness of our meta language CIC in enabling succinct definitions.

With these relations in hand, we reach the point where most pencil-and-paper proofs would say that the rest follows by "routine inductions" of appropriate kinds. Unfortunately, since we want to convince the Coq proof checker, we will need to provide considerably more detail. There is no magic bullet for automating proofs of this sophistication, but I did identify some techniques that worked surprisingly well for simplifying the proof burden. In particular, since this compiler preserves type information, I was able to automate significant portions of the proofs using type-based heuristics. The key insight was the possibility of using *greedy quantifier instantiation*, since the dependent types I use are so particular that most logical quantifiers have domains compatible with just a single subterm of a proof sequent.

```
Inductive lprimop : list type → type → Set :=
  | LConst : ∀Γ. ℕ → lprimop Γ Nat
  | LVar : ∀Γ. ∀τ. var Γ τ → lprimop Γ τ
  | LLam : ∀Γ. ∀τ1. ∀τ2. lterm (τ1 :: Γ) τ2 → lprimop Γ (Arrow τ1 τ2)
 with lterm : list type → type → Set :=
  | LLet : ∀Γ. ∀τ1. ∀τ2. lprimop Γ τ1 → lterm (τ1 :: Γ) τ2 → lterm Γ τ2
  | LThrow : ∀Γ. ∀τ. lprimop Γ τ → lterm Γ τ
  | LApp : ∀Γ. ∀τ1. ∀τ2. ∀τ3.
    var Γ (Arrow τ1 τ2) → var Γ τ1 → var Γ (Arrow τ2 τ3) → lterm Γ τ3
```

Figure 7.15: Coq definition of syntax and static semantics for Linear

7.4 Representing Typed Languages

We begin our example by representing the target language of its transformation. The first step is easy; we give in Figure 7.14 a standard algebraic datatype definition of the type language shared by Source and Linear. This is the same definition that we gave in the last chapter for simply-typed lambda calculus. Defining Linear's operands and terms is a bit more interesting, as we must define them as two *mutually-inductive* Coq types. Figure 7.15 provides the details. For reference, Figure 7.4 specified Linear's syntax in a more traditional style.

To give an idea of the make-up of the real implementation, I will also provide some concrete Coq code snippets throughout this chapter. The syntax and static and dynamic semantics of our source language are simple enough that I can show them here in their entirety, in Figure 7.16. Though the ASCII Coq notation introduces some complication, the structure of these definitions really mirrors their informal counterparts exactly. These snippets are only meant to give a flavor of the project. I describe in Section 7.10 how to obtain the complete project source code, for those who want a more in-depth treatment.

```
Inductive sty : Set :=
  | SNat : sty
  | SArrow : sty -> sty -> sty.

Inductive sterm : list sty -> sty -> Set :=
  | SVar : forall G t,
    Var G t
    -> sterm G t
  | SLam : forall G dom ran,
    sterm (dom :: G) ran
    -> sterm G (SArrow dom ran)
  | SApp : forall G dom ran,
    sterm G (SArrow dom ran)
    -> sterm G dom
    -> sterm G ran
  | SConst : forall G, nat -> sterm G SNat.

Fixpoint styDenote (t : sty) : Set :=
  match t with
    | SNat => nat
    | SArrow t1 t2 => styDenote t1 -> styDenote t2
  end.

Fixpoint stermDenote (G : list sty) (t : sty)
    (e : sterm G t) {struct e}
  : Subst styDenote G -> styDenote t :=
  match e in (sterm G t)
      return (Subst styDenote G -> styDenote t) with
    | SVar _ _ v => fun s =>
      VarDenote v s
    | SLam _ _ _ e' => fun s =>
      fun x => stermDenote e' (SCons x s)
    | SApp _ _ _ e1 e2 => fun s =>
      (stermDenote e1 s) (stermDenote e2 s)
    | SConst _ n => fun _ => n
  end.
```

Figure 7.16: Coq source code of Source syntax and semantics

$$\begin{array}{rcl}
(\textsf{let } y = o \textsf{ in } e_1) \overset{u}{\bullet} e_2 & = & \textsf{let } y = o \textsf{ in } (e_1 \overset{u}{\bullet} e_2) \\
(\textsf{throw } \langle o \rangle) \overset{u}{\bullet} e & = & \textsf{let } u = o \textsf{ in } e \\
(x\ y\ z) \overset{u}{\bullet} e & = & \textsf{let } f = (\lambda v.\ \textsf{let } g = (\lambda u.\ e) \textsf{ in } z\ v\ g) \\
& & \textsf{in } x\ y\ f
\end{array}$$

Figure 7.17: Continuation composition operator

$$\begin{array}{rcl}
\lfloor n \rfloor & = & \textsf{throw } \langle n \rangle \\
\lfloor x \rfloor & = & \textsf{throw } \langle x \rangle \\
\lfloor e_1\ e_2 \rfloor & = & \lfloor e_1 \rfloor \overset{u}{\bullet} (\lfloor e_2 \rfloor \overset{v}{\bullet} (\textsf{let } f = (\lambda x.\ \textsf{throw } \langle x \rangle) \textsf{ in } u\ v\ f)) \\
\lfloor \lambda x : \tau.\ e \rfloor & = & \textsf{throw } \langle \lambda x : \tau.\ \lfloor e \rfloor \rangle
\end{array}$$

Figure 7.18: Linearization translation

7.5 Representing Transformations

We are able to write the linearization transformation quite naturally using our de Bruijn terms. I start this section with a less formal presentation of it.

I first define in Figure 7.17 an auxiliary operation $e_1 \overset{u}{\bullet} e_2$ that, given two linear terms e_1 and e_2 and a variable u free in e_2, returns a new linear term equivalent to running e_1 and throwing its result to e_2 by binding that result to u.

Now we can give the linearization translation itself, as shown in Figure 7.18. This translation can be converted rather directly into Coq recursive function definitions. The catch comes as a result of our strong dependent types for program syntax. The Coq type checker is not able to verify the type-correctness of some of the clauses above.

For a simple example, let us focus on part of the linearization case for applications. There we produce terms of the form $\lfloor e_1 \rfloor \overset{u}{\bullet} (\lfloor e_2 \rfloor \overset{v}{\bullet} \ldots)$. The term e_2 originally occurred in some context Γ. However, here $\lfloor e_2 \rfloor$, the compilation of e_2, is used in a context formed by adding an additional variable u to Γ. We know that this transplantation is harmless and ought to be allowed, but the Coq type checker flags this section as a type error.

We need to perform an explicit coercion that adjusts the de Bruijn indices in $\lfloor e_2 \rfloor$. The operation we want corresponds to a weakening lemma for our typing judgment: "If $\Gamma \vdash e : \tau$, then $\Gamma, x : \tau' \vdash e : \tau$ when x is not free in e." Translating this description into our formalization with inductive types from the last section, we want a function:

$$\mathsf{weakenFront} \quad : \quad \forall \Gamma.\ \forall \tau.\ \mathsf{lterm}\ \Gamma\ \tau \rightarrow \forall \tau'.\ \mathsf{lterm}\ (\tau' :: \Gamma)\ \tau$$

It is possible to write a custom weakening function for linearized terms, but nothing about the implementation is specific to our language. There is a generic recipe for building weakening functions, based only on an understanding of where variable binders occur. To keep the translations free of such details, I have opted to create a generic programming system that builds these syntactic helper functions for us. The next section introduces it.

When we insert these coercions where needed, we have a direct translation of our informal compiler definition into Coq code. Assuming for now that the coercion functions have already been generated, I can give as an example of a real implementation the Coq code for the main CPS translation, in Figure 7.19. As with programming in general, the code contains details that can obscure the meaning at first, but there really isn't that much going on. I recall here that the `Next` and `First` constructors are used to build de Bruijn variables in unary form. `compose` is the name of the function corresponding to our informal $\stackrel{u}{\bullet}$ operator. `Lterm` is a module of helper functions generated automatically, and it includes the coercion `weakenFront` described earlier. Let us, then, turn to a discussion of the generic

```
Fixpoint cps (G : list sty) (t : sty)
    (e : sterm G t) {struct e}
  : lterm G t :=
  match e in (sterm G t) return (lterm G t) with
    | SVar _ _ v => LThrow (LVar v)
    | SConst _ n => LThrow (LConst _ n)
    | SLam _ _ _ e' => LThrow (LLam (cps e'))
    | SApp _ _ _ e1 e2 =>
      compose (cps e1)
      (compose (Lterm.weakenFront _ (cps e2))
         (LBind
          (LLam (LThrow (LVar First)))
          (LApply
            (Next (Next First))
            (Next First)
            First)))
  end.
```

Figure 7.19: Coq source code of the main CPS translation

programming system that produces it.

7.6 Generic Syntactic Functions

A variety of these syntactic helper functions come up again and again in formalization of programming languages. I have identified a few primitive functions that seemed to need per-language implementations, along with a number of derived functions that can be implemented parametrically in the primitives. My generic programming system writes the primitive functions for the user and then automatically instantiates the parametric derived functions. I present here a catalogue of the functions that I found to be important in this present work. All operate over some type family `term` parameterized by types from an arbitrary language.

The first primitive is a generalization of the weakening function from the last section. I modify its specification to allow insertion into any position of a context, not just the beginning. $\oplus$ denotes list concatenation, and it and single-element concatenation :: are right associative at the same precedence level.

$$\mathsf{weaken} \quad : \quad \forall \Gamma_1.\ \forall \Gamma_2.\ \forall \tau.\ \mathsf{term}\ (\Gamma_1 \oplus \Gamma_2)\ \tau \to \forall \tau'.\ \mathsf{term}\ (\Gamma_1 \oplus \tau' :: \Gamma_2)\ \tau$$

Next is elementwise permutation. We swap the order of two adjacent context elements.

$$\mathsf{permute} \quad : \quad \forall \Gamma_1.\ \forall \Gamma_2.\ \forall \tau_1.\ \forall \tau_2.\ \forall \tau.\ \mathsf{term}\ (\Gamma_1 \oplus \tau_1 :: \tau_2 :: \Gamma_2)\ \tau \to \mathsf{term}\ (\Gamma_1 \oplus \tau_2 :: \tau_1 :: \Gamma_2)\ \tau$$

We also want to calculate the free variables of a term. Here I mean those that actually appear, not just those that are in scope. This calculation and related support functions are critical to efficient closure conversion for any language. I write $\mathcal{P}(\Gamma)$ to denote the type of subsets of the bindings in context Γ.

$$\mathsf{freeVars} \quad : \quad \forall \Gamma.\ \forall \tau.\ \mathsf{term}\ \Gamma\ \tau \to \mathcal{P}(\Gamma)$$

Using the notion of free variables, we can define *strengthening*, which removes unused variable bindings from a context. This operation is also central to closure conversion. An argument of type $\Gamma_1 \subseteq \Gamma_2$ denotes a constructive proof that every binding in Γ_1 is also in Γ_2.

$$\mathsf{strengthen} \quad : \quad \forall \Gamma.\ \forall \tau.\ \forall e : \mathsf{term}\ \Gamma\ \tau.\ \forall \Gamma'.\ \mathsf{freeVars}\ \Gamma\ \tau\ e \subseteq \Gamma' \to \mathsf{term}\ \Gamma'\ \tau$$

I have found these primitive operations to be sufficient for easing the burden of manipulating the higher-level intermediate languages. The derived operations that the system instantiates are: adding multiple bindings to the middle of a context and adding multiple bindings to the end of a context, based on weaken; and moving a binding from the front to the middle or from the middle to the front of a context and swapping two adjacent multi-binding sections of a context, derived from $\mathsf{permute}$.

7.6.1 Generic Correctness Proofs

Writing these boilerplate syntactic functions is less than half of the challenge when it comes to proving semantics preservation. The semantic properties of these functions are crucial to enabling correctness proofs for the transformations that use them. I also automate the generation of the correctness proofs, based on an observation about the classes of semantic properties we find ourselves needing to verify for them. The key insight is that we only require that each function *commute with any compositional denotation function* in a function-specific way.

An example should illustrate this idea best. Take the case of the $\mathsf{weakenFront}$ function for our Source language. Fix an arbitrary denotation function $[\![\cdot]\!]$ for source terms. I claim that the correctness of a well-written compiler will only depend on the following property of $\mathsf{weakenFront}$, written with informal notation: For any Γ, τ, and e with $\Gamma \vdash e : \tau$, we have for any τ', substitution σ for the variables of Γ, and value v of type $[\![\tau']\!]$, that:

$$[\![\mathsf{weakenFront}\ \Gamma\ \tau\ e\ \tau']\!](\sigma, v) = [\![e]\!]\sigma$$

We shouldn't need to know many specifics about $[\![\cdot]\!]$ to deduce this theorem. In fact, it turns out that all we need is the standard property used to judge suitability of denotational semantics, compositionality. In particular, we require that there exist functions

f_{const}, f_{var}, f_{app}, and f_{lam} such that:

$$\begin{array}{rcl}
[\![n]\!]\sigma & = & f_{const}(n) \\
[\![x]\!]\sigma & = & f_{var}(\sigma(x)) \\
[\![e_1\ e_2]\!]\sigma & = & f_{app}([\![e_1]\!]\sigma, [\![e_2]\!]\sigma) \\
[\![\lambda x : \tau.\ e]\!]\sigma & = & f_{lam}(\lambda x : [\![\tau]\!].\ [\![e]\!](\sigma, x))
\end{array}$$

My generic programming system introspects into user-supplied denotation function definitions and extracts the functions that witness their compositionality. Using these functions, it performs automatic proofs of generic theorems like the one above about weakenFront. Every other generated syntactic function has a similar customized theorem statement and automatic strategy for proving it for compositional denotations.

The means of achieving this functionality are interesting in their own right as an approach to generic programming and proving, so I will describe them at length in Chapter 8.

7.7 Representing Logical Relations

We now turn our attention to formulating the correctness proofs of compiler phases, again using the linearization phase as our example. We are able to give a very simple logical relations argument for this phase, since our meta language CIC is sufficiently expressive to encode naturally the features of the source and target languages. The correctness theorem that we want looks something like the following, for some appropriate type-indexed relation $\simeq$. For disambiguation purposes, I write $[\![\cdot]\!]^S$ for Source language denotations and $[\![\cdot]\!]^L$ for Linear denotations.

Theorem 3 (Correctness of linearization) ***For*** *Source term e such that $\Gamma \vdash e : \tau$,* ***if*** *we have substitutions σ^S and σ^L for $[\![\Gamma]\!]^S$ and $[\![\Gamma]\!]^L$, respectively,* ***such that*** *for every $x : \tau' \in \Gamma$, we have $\sigma^S(x) \simeq_{\tau'} \sigma^L(x)$,* ***then*** *$[\![e]\!]^S \sigma^S \simeq_\tau [\![\lfloor e \rfloor]\!]^L \sigma^L$.*

This relation for values turns out to satisfy our requirements:

$$\begin{array}{rcl} n_1 \simeq_{\mathsf{N}} n_2 & = & n_1 = n_2 \\ f_1 \simeq_{\tau_1 \to \tau_2} f_2 & = & \forall x_1 : [\![\tau_1]\!]^S, \forall x_2 : [\![\tau_1]\!]^L, x_1 \simeq_{\tau_1} x_2 \\ & & \to \exists v : [\![\tau_2]\!]^L, \forall k : [\![\tau_2]\!]^L \to \mathbb{N}, \\ & & \ f_2 \ x_2 \ k = k \ v \land f_1 \ x_1 \simeq_{\tau_2} v \end{array}$$

We have a standard logical relation defined by recursion on the structure of types. $e_1 \simeq_\tau e_2$ means that values e_1 of type $[\![\tau]\!]^S$ and e_2 of type $[\![\tau]\!]^L$ are equivalent in a suitable sense. Numbers are equivalent if and only if they are equal. Source function f_1 and linearized function f_2 are equivalent if and only if for every pair of arguments x_1 and x_2 related at the domain type, there exists some value v such that f_2 called with x_2 and a continuation k always throws v to k, and the result of applying f_1 to x_1 is equivalent to v at the range type.

The suitability of particular logical relations to use in compiler phase correctness specifications is hard to judge individually. We know we have made proper choices when we are able to compose all of our correctness results to form the overall compiler correctness theorem. It is probably also worth pointing out here that my denotational semantics are not *fully abstract*, in the sense that the target domains "allow more behavior" than the object languages ought to. For instance, the functions k quantified over by the function case of the $\simeq$ definition are drawn from the full Coq function space, which includes all manner of complicated functions relying on inductive and co-inductive types. The acceptability of this

```
Fixpoint val_lr (t : sty) : styDenote t -> ltyDenote t -> Prop :=
  match t
    return (styDenote t -> ltyDenote t -> Prop) with
    | SNat => fun n1 n2 =>
      n1 = n2
    | SArrow t1 t2 => fun f1 f2 =>
      forall x1 x2, val_lr t1 x1 x2
        -> forall (k : styDenote t2 -> nat),
          exists thrown, f2 x2 k = k thrown
            /\ val_lr t2 (f1 x1) thrown
  end.
```

Figure 7.20: Coq source code for the first-stage CPS transform's logical relation

choice for this application is borne out by my success in using this logical relation to prove a final theorem, whose statement does not depend on such quantifications.

Once we are reconciled with that variety of caveat, we find that Coq provides quite a congenial platform for defining logical relations for denotational semantics. The $\simeq$ definition can be transcribed quite literally, as witnessed by Figure 7.20. The set of all logical propositions in Coq is just another type Prop, and so we may write recursive functions that return values in it. Contrast our options here with those associated with proof assistants like Twelf [76], for which formalization of logical relations has historically been challenging.

7.8 Proof Automation

Now that we have the statement of our theorem, we need to produce a formal proof of it. In general, our proofs will require significant manual effort. As anyone who has worked on computerized proofs can tell you, time-saving automation machinery is extremely welcome. In proving the correctness theorems for this compiler, I was surprised to find that a very simple automation technique is very effective on large classes of proof goals that appear. The effectiveness of this technique has everything to do with the combination of typed intermediate languages and use of dependent types to represent their programs.

As an example, consider this proof obligation that occurs in the correctness proof for linearization. It is the induction step associated with function applications.

$$
\begin{array}{rcl}
H_1 & : & \forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_\Gamma \sigma^L \to \exists v : [\![\tau_1 \to \tau_2]\!]^L, \\
 & & \forall k : [\![\tau_1 \to \tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1 \rfloor]\!]^L \sigma^L \ k = k \ v \wedge [\![e_1]\!]^S \sigma^S \simeq_{\tau_1 \to \tau_2} v \\
H_2 & : & \forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_\Gamma \sigma^L \to \exists v : [\![\tau_1]\!]^L, \\
 & & \forall k : [\![\tau_1]\!]^L \to \mathbb{N}, [\![\lfloor e_2 \rfloor]\!]^L \sigma^L \ k = k \ v \wedge [\![e_2]\!]^S \sigma^S \simeq_{\tau_1} v \\
\hline
 & & \forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_\Gamma \sigma^L \to \exists v : [\![\tau_2]\!]^L, \\
 & & \forall k : [\![\tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1 \ e_2 \rfloor]\!]^L \sigma^L \ k = k \ v \wedge [\![e_1 \ e_2]\!]^S \sigma^S \simeq_{\tau_2} v
\end{array}
$$

This is an example of a Coq proof-search sequent. Above the horizontal line, we have the named *hypotheses* H_1 and H_2. Below the line, we have the *conclusion*. We are trying to deduce the conclusion from the hypotheses.

It is safe to simplify the conclusion by moving all of the universal quantifiers and implications that begin it into our proof context as new bound variables and hypotheses; this rearrangement cannot alter the provability of the goal.

$$
\begin{array}{rcl}
H_1 & : & \forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_\Gamma \sigma^L \to \exists v : [\![\tau_1 \to \tau_2]\!]^L, \\
 & & \forall k : [\![\tau_1 \to \tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1 \rfloor]\!]^L \sigma^L \ k = k \ v \wedge [\![e_1]\!]^S \sigma^S \simeq_{\tau_1 \to \tau_2} v \\
H_2 & : & \forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_\Gamma \sigma^L \to \exists v : [\![\tau_1]\!]^L, \\
 & & \forall k : [\![\tau_1]\!]^L \to \mathbb{N}, [\![\lfloor e_2 \rfloor]\!]^L \sigma^L \ k = k \ v \wedge [\![e_2]\!]^S \sigma^S \simeq_{\tau_1} v \\
\sigma^S & : & \ldots \\
\sigma^L & : & \ldots \\
H_3 & : & \sigma^S \simeq_\Gamma \sigma^L \\
\hline
 & & \exists v : [\![\tau_2]\!]^L, \\
 & & \forall k : [\![\tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1 \ e_2 \rfloor]\!]^L \sigma^L \ k = k \ v \wedge [\![e_1 \ e_2]\!]^S \sigma^S \simeq_{\tau_2} v
\end{array}
$$

Beyond that, Coq's standard automation support is stuck. However, it turns out that we can do much better for goals like this one, based on *greedy quantifier instantiation*. Traditional automated theorem provers spend most of their intelligence in determining how to *use* universally-quantified facts and *prove* existentially-quantified facts. When quantifiers range over infinite domains, many such theorem-proving problems are both undecidable and difficult in practice.

However, examining our goal above, we notice that it has a very interesting property: *every* quantifier has a rich type depending on some object language type. Moreover, for any of these types, *exactly one subterm of proof state* (bound variables, hypotheses, and conclusion) that has that type ever appears at any point during the proving process! This observation makes instantiation of quantifiers extremely easy: instantiate any universal hypothesis or existential conclusion with *the first properly-typed proof state subterm that appears.*

For instance, in this example, we had just moved the variables σ^S and σ^L and the assumption $\sigma^S \simeq_\Gamma \sigma^L$ into our proof context. That means that we should instantiate the initial σ quantifiers of H_1 with these variables and use modus ponens with H_3 to access the conclusion of H_1's implication.

$$
\begin{array}{rcl}
H_1 & : & \exists v : [\![\tau_1 \to \tau_2]\!]^L, \\
 & & \forall k : [\![\tau_1 \to \tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_1]\!]^S \sigma^S \simeq_{\tau_1 \to \tau_2} v \\
H_2 & : & \forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_\Gamma \sigma^L \to \exists v : [\![\tau_1]\!]^L, \\
 & & \forall k : [\![\tau_1]\!]^L \to \mathbb{N}, [\![\lfloor e_2 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_2]\!]^S \sigma^S \simeq_{\tau_1} v \\
\sigma^S & : & \ldots \\
\sigma^L & : & \ldots \\
H_3 & : & \sigma^S \simeq_\Gamma \sigma^L \\
\hline
 & & \exists v : [\![\tau_2]\!]^L, \\
 & & \forall k : [\![\tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1\ e_2 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_1\ e_2]\!]^S \sigma^S \simeq_{\tau_2} v
\end{array}
$$

This operation leaves us at an existential quantifier, so we eliminate that quantifier by adding a fresh variable v_1 and a new assumption about that variable.

$$
\begin{array}{rcl}
v_1 & : & [\![\tau_1 \to \tau_2]\!]^L \\
H_1 & : & \forall k : [\![\tau_1 \to \tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1 \rfloor]\!]^L \sigma^L\ k = k\ v_1 \wedge [\![e_1]\!]^S \sigma^S \simeq_{\tau_1 \to \tau_2} v_1 \\
H_2 & : & \forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_\Gamma \sigma^L \to \exists v : [\![\tau_1]\!]^L, \\
 & & \forall k : [\![\tau_1]\!]^L \to \mathbb{N}, [\![\lfloor e_2 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_2]\!]^S \sigma^S \simeq_{\tau_1} v \\
\sigma^S & : & \ldots \\
\sigma^L & : & \ldots \\
H_3 & : & \sigma^S \simeq_\Gamma \sigma^L \\
\hline
 & & \exists v : [\![\tau_2]\!]^L, \\
 & & \forall k : [\![\tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1\ e_2 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_1\ e_2]\!]^S \sigma^S \simeq_{\tau_2} v
\end{array}
$$

The type of the k quantifier that we reach now don't match any subterms in scope, so we stop here. However, we can repeat the process on H_2:

$$\begin{array}{rcl}
v_1 & : & [\![\tau_1 \to \tau_2]\!]^L \\
H_1 & : & \forall k : [\![\tau_1 \to \tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1 \rfloor]\!]^L \sigma^L \ k = k \ v_1 \wedge [\![e_1]\!]^S \sigma^S \simeq_{\tau_1 \to \tau_2} v_1 \\
v_2 & : & [\![\tau_1]\!]^L \\
H_2 & : & \forall k : [\![\tau_1]\!]^L \to \mathbb{N}, [\![\lfloor e_2 \rfloor]\!]^L \sigma^L \ k = k \ v_2 \wedge [\![e_2]\!]^S \sigma^S \simeq_{\tau_1} v_2 \\
\sigma^S & : & \ldots \\
\sigma^L & : & \ldots \\
H_3 & : & \sigma^S \simeq_\Gamma \sigma^L \\
\hline
 & & \exists v : [\![\tau_2]\!]^L, \\
 & & \forall k : [\![\tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1 \ e_2 \rfloor]\!]^L \sigma^L \ k = k \ v \wedge [\![e_1 \ e_2]\!]^S \sigma^S \simeq_{\tau_2} v
\end{array}$$

Now it's time for a round of rewriting, using rules added by the human user to a hint database. We use all of the boilerplate syntactic function soundness theorems that we generated automatically as left-to-right rewrite rules, applying them in the goal until no further changes are possible. Using also a rewrite theorem expressing the soundness of the $\overset{u}{\bullet}$ composition operation, this process simplifies the goal to a form with a subterm $[\![\lfloor e_1 \rfloor]\!]^L \ \sigma^L \ (\lambda x : [\![\tau_1 \to \tau_2]\!]^L. \ ...)$:

$$\begin{array}{rcl}
v_1 & : & [\![\tau_1 \to \tau_2]\!]^L \\
H_1 & : & \forall k : [\![\tau_1 \to \tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1 \rfloor]\!]^L \sigma^L \ k = k \ v_1 \wedge [\![e_1]\!]^S \sigma^S \simeq_{\tau_1 \to \tau_2} v_1 \\
v_2 & : & [\![\tau_1]\!]^L \\
H_2 & : & \forall k : [\![\tau_1]\!]^L \to \mathbb{N}, [\![\lfloor e_2 \rfloor]\!]^L \sigma^L \ k = k \ v_2 \wedge [\![e_2]\!]^S \sigma^S \simeq_{\tau_1} v_2 \\
\sigma^S & : & \ldots \\
\sigma^L & : & \ldots \\
H_3 & : & \sigma^S \simeq_\Gamma \sigma^L \\
\hline
 & & \exists v : [\![\tau_2]\!]^L, \\
 & & \forall k : [\![\tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1 \rfloor]\!]^L \ \sigma^L \ (\lambda x : [\![\tau_1 \to \tau_2]\!]^L, ...) = \ldots \wedge \ldots
\end{array}$$

This lambda term has the right type to use as an instantiation for the universally-quantified k in H_1, so, by our greedy heuristic, we make that instantiation.

$$
\begin{array}{rcl}
v_1 & : & [\![\tau_1 \rightarrow \tau_2]\!]^L \\
H_1 & : & [\![\lfloor e_1 \rfloor]\!]^L \sigma^L \ (\lambda x : [\![\tau_1 \rightarrow \tau_2]\!]^L, ...) = (\lambda x : [\![\tau_1 \rightarrow \tau_2]\!]^L, ...) \ v_1 \wedge \ldots \\
v_2 & : & [\![\tau_1]\!]^L \\
H_2 & : & \forall k : [\![\tau_1]\!]^L \rightarrow \mathbb{N}, [\![\lfloor e_2 \rfloor]\!]^L \sigma^L \ k = k \ v_2 \wedge [\![e_2]\!]^S \sigma^S \simeq_{\tau_1} v_2 \\
\sigma^S & : & \ldots \\
\sigma^L & : & \ldots \\
H_3 & : & \sigma^S \simeq_\Gamma \sigma^L \\
\hline
 & & \exists v : [\![\tau_2]\!]^L, \\
 & & \forall k : [\![\tau_2]\!]^L \rightarrow \mathbb{N}, [\![\lfloor e_1 \rfloor]\!]^L \ \sigma^L (\lambda x : [\![\tau_1 \rightarrow \tau_2]\!]^L, ...) = \ldots \wedge \ldots
\end{array}
$$

Now H_1 states an equality that can be used to rewrite the conclusion.

$$
\begin{array}{rcl}
v_1 & : & [\![\tau_1 \rightarrow \tau_2]\!]^L \\
H_1 & : & [\![\lfloor e_1 \rfloor]\!]^L \sigma^L \ (\lambda x : [\![\tau_1 \rightarrow \tau_2]\!]^L, ...) = (\lambda x : [\![\tau_1 \rightarrow \tau_2]\!]^L, ...) \ v_1 \wedge \ldots \\
v_2 & : & [\![\tau_1]\!]^L \\
H_2 & : & \forall k : [\![\tau_1]\!]^L \rightarrow \mathbb{N}, [\![\lfloor e_2 \rfloor]\!]^L \sigma^L \ k = k \ v_2 \wedge [\![e_2]\!]^S \sigma^S \simeq_{\tau_1} v_2 \\
\sigma^S & : & \ldots \\
\sigma^L & : & \ldots \\
H_3 & : & \sigma^S \simeq_\Gamma \sigma^L \\
\hline
 & & \exists v : [\![\tau_2]\!]^L, \\
 & & \forall k : [\![\tau_2]\!]^L \rightarrow \mathbb{N}, (\lambda x : [\![\tau_1 \rightarrow \tau_2]\!]^L, ...) \ v_1 = \ldots \wedge \ldots
\end{array}
$$

When we continue iterating this heuristic, we discharge the proof obligation completely, with no human intervention. Our very naive algorithm has succeeded in "guessing" all of the complicated continuations needed for quantifier instantiations, simply by searching for properly-typed subterms of the proof state. In fact, this same heuristic discharges all of the cases of the linearization soundness proof, once we've stocked the rewrite database with the appropriate syntactic simplifications beforehand. To prove this theorem, all the user needs to do is specify which induction principle to use and run the heuristic on the resulting subgoals.

I have found this approach to be very effective on high-level typed languages in general. The human prover's job is to determine the useful syntactic properties with which to augment those proved generically, prove these new properties, and add them to the rewrite hint database. With very little additional effort, the main theorems can then be discharged automatically. Unfortunately, the lower-level intermediate languages keep less precise type information, so greedy instantiation would make incorrect choices too often to be practical. Thus, my experience here provides a new justification in support of type-preserving compilation.

Figure 7.21 shows example Coq code for proving the CPS correctness theorem, with a few small simplifications made for clarity.

7.9 Further Discussion

Some other aspects of my formalization outside of the running example are worth mentioning. I summarize them in this section.

7.9.1 Logical Relations for Closure Conversion

Formalizations of closure conversion and its correctness, especially those using operational semantics, often involve existential types and other relatively complicated notions. In my formalization, I am able to use a surprisingly simple logical relation to characterize closure conversion. It relates denotations from the CPS and CC languages, indicated with superscripts P and C, respectively. I lift various definitions to vectors in the usual way.

```
(* State the lemma characterizing the effect of
 * the CPS'd term composition operator. *)
Lemma compose_sound : forall (G : list sty)
  (t : sty) (e : lterm G t)
  (t' : sty) (e' : lterm (t :: G) t') s
  (k : _ -> result),
  ltermDenote (compose e e') s k
  = ltermDenote lin s
    (fun x => ltermDenote lin' (SCons x s) k).
Proof.
  induction e;    (* We prove it by induction on
                   * the structure of
                   * the term e. *)
    equation_tac. (* A generic rewriting
                   * procedure handles
                   * the resulting cases. *)
Qed.

(* ...omitted code to add compose_sound to our
 * rewriting hint base... *)

(* State the theorem characterizing soundness of
 * the CPS translation. *)
Theorem cps_sound : forall G t (e : sterm G t),
  exp_lr e (cps e).
Proof.
  unfold exp_lr; (* Expand the definition of the
                  * logical relation on
                  * expressions. *)
    induction e; (* Proceed by induction on the
                  * structure of the term e. *)
    lr_tac.      (* Use the generic greedy
                  * instantiation procedure to
                  * discharge the subgoals. *)
Qed.
```

Figure 7.21: Snippets of the Coq proof script for the CPS correctness theorem

$$\begin{array}{rcl} n_1 \simeq_{\mathsf{N}} n_2 & = & n_1 = n_2 \\ f_1 \simeq_{\vec{\tau} \to \mathbb{N}} f_2 & = & \forall \vec{x}_1 : [\![\vec{\tau}]\!]^P, \forall x_2 : [\![\vec{\tau}]\!]^C, x_1 \simeq_{\vec{\tau}} x_2 \\ & & \to f_1\ x_1 \simeq_{\tau_2} f_2\ x_2 \end{array}$$

This relation is almost identical to the most basic logical relation for simply-typed lambda calculus! The secret is that our meta language CIC "has native support for closures." That is, Coq's function spaces already incorporate the appropriate reduction rules to capture free variables, so we don't need to mention this process explicitly in our relation.

In more detail, the relevant denotations of CC types are:

$$\begin{array}{rcl} [\![\vec{\tau} \to \mathbb{N}]\!] & = & [\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!] \to \mathbb{N} \\ [\![\vec{\tau}^1 \times \vec{\tau}^2 \to \mathbb{N}]\!] & = & ([\![\tau^1_1]\!] \times \ldots \times [\![\tau^1_n]\!]) \to ([\![\tau^2_1]\!] \times \ldots \times [\![\tau^2_m]\!]) \to \mathbb{N} \end{array}$$

Recall that the second variety of type shown is the type of code pointers, with the additional first list of parameters denoting the expected environment type. A CPS language lambda expression is compiled into a packaging of a closure using a pointer to a fresh code block. That code block will have some type $\vec{\tau}_1 \times \vec{\tau}_2 \to \mathbb{N}$. The denotation of this type is some meta language type $T_1 \to T_2 \to \mathbb{N}$. We perform an immediate partial application to an environment formed from the relevant free variables. This environment's type will have denotation T_1. Thus, the partial application's type has denotation $T_2 \to \mathbb{N}$, making it compatible with our logical relation. The effect of the closure packaging operation has been "hidden" using one of the meta language's "closures" formed by the partial application.

7.9.2 Explicating Higher-Order Control Flow

The translation from CC to Alloc moves from a higher-order, terminating language to a first-order language that admits non-termination. As a consequence, the translation correctness proof must correspond to some explanation of why CC's particular brand of higher-order control flow leads to termination, and why the resulting first-order program returns an identical answer to the higher-order original.

The basic proof technique has two main pieces. First, the logical relation requires that any value with a code type terminates with the expected value when started in a heap and variable environment where the same condition holds for values that should have code type. Second, I take advantage of the fact that function definitions occur in dependency order in CC programs. Our variable binding restrictions enforce a lack of cycles via dependent types. We can prove by induction on the position of a code body in a program that any execution of it in a suitable starting state terminates with the correct value. Any code pointer involved in the execution either comes earlier in the program, in which case we have its correctness by the inductive hypothesis; or the code pointer has been retrieved from a variable or heap slot, in which case its safety follows by an induction starting from our initial hypothesis and tracking the variable bindings and heap writes made by the program.

7.9.3 Garbage Collection Safety

The soundness of the translation from Flat to Asm depends on a delicate safety property of well-typed Flat programs. It is phrased in terms of the register typings we have available at every program point, and it depends on the definition of pointer isomorphism

I gave in Section 7.2.7.

Theorem 4 (Heap rearrangement safety) *For any register typing Δ, abstract heaps m_1 and m_2, and register files R_1 and R_2, if:*

1. *For every register r with $\Delta(r) = \mathsf{ref}$, $R_1(r)$ is isomorphic to $R_2(r)$ with respect to m_1 and m_2.*

2. *For every register r with $\Delta(r) \neq \mathsf{ref}$, $R_1(r) = R_2(r)$.*

and p is a Flat program such that $\Delta \vdash p$, we have $[\![p]\!]R_1m_1 = [\![p]\!]R_2m_2$.

This theorem says that it is safe to rearrange a heap if the relevant roots pointing into it are also rearranged equivalently. Well-typed Flat programs can't distinguish between the old and new situations. The denotations that we conclude are equal in the theorem are traces, so we have that valid Flat programs return identical results (if any) and make the same numbers of function calls in any isomorphic initial states.

The requirements on the runtime system's new operation allow it to modify the heap arbitrarily on each call, so long as the new heap and registers are isomorphic to the old heap and registers. The root set provided as an operand to new is used to determine the appropriate notion of isomorphism, so the heap rearrangement safety theorem is critical to making the correctness proof go through.

7.9.4 Putting It All Together

With all of the compiler phases proved, we can compose the proofs to form a correctness proof for the overall compiler. I give the formal version of the theorem described informally in the Introduction. I use the notation $T \downarrow n$ to denote that trace T terminates with result n.

Theorem 5 (Compiler correctness) *Let $m \in \mathbb{H}$ be a heap initialized with a closure for the top-level continuation, let p be a pointer to that closure, and let R be a register file mapping the first register to p. Given a Source term e such that $\cdot \vdash e : \mathsf{N}$, $[\![\lfloor e \rfloor]\!] Rm \downarrow [\![e]\!]()$.*

Even considering only the early phases of the compiler, it is difficult to give a correctness theorem in terms of equality for all source types. For instance, we can't compose parametrically the results for CPS transformation and closure conversion to yield a correctness theorem expressed with an equality between denotations. The problem lies in the lack of full abstraction for our semantics. Instead, we must use an alternate notion of equality that only requires functions to agree at arguments that are denotations of terms. This general notion also appears in the idea of pre-logical relations [47], a compositional alternative to logical relations.

7.10 Implementation

The compiler implementation and documentation are available online at:

`http://ltamer.sourceforge.net/`

I present lines-of-code counts for the different implementation files, including proofs, in Figure 7.22. I divide the files between those that define languages and their semantics, in the first column and every other column thereafter; and those files that formalize translations between languages and their correctness proofs, situated between the

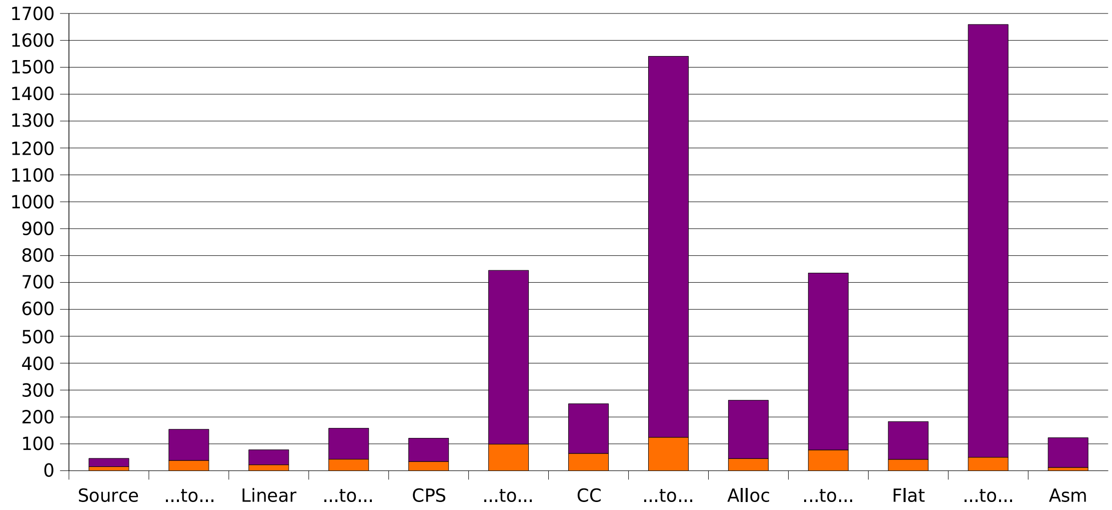

Figure 7.22: Project lines-of-code counts

columns for their source and target languages. Each bar shows how many lines of code that component encompasses. Additionally, each bar has a portion of its bottom marked off, indicating my estimate of how many of the lines of code would remain in a traditional ML implementation of that piece. The change from the the bottom part of a bar to the bar's full height indicates roughly how hard it was to certify that component.

Not surprisingly, the two bars exhibiting the highest certification overheads are those for the two translations between the most dissimilar language pairs. The translation from CC to Alloc goes from higher-order to first-order handling of code, and the translation from Flat to Asm goes from an abstract heap to a concrete heap with garbage collection. The overall certification overhead for the project stands at around a factor of 10.

The graph does not include 3520 lines of Coq code from the Lambda Tamer library, which I developed in tandem with the compiler. Additionally, that library contains 2716 lines of OCaml code supporting generic programming of syntactic support functions and their correctness proofs. Developing these re-usable pieces was the most time-consuming part of the overall effort. I believe I improved my productivity an order of magnitude by determining the right language representation scheme and its library support code, and again by developing the generic programming system. With those pieces in place, implementing the compiler only took about one month of work, which I feel validates the general efficacy of my techniques.

It is worth characterizing how much of the implementation must be trusted to trust its outputs. Of course, the Coq system and the toolchain used to compile its programs (including operating system and hardware) must be trusted. Focusing on code specific to this project, we see that, if we want only to certify the behavior of particular output assembly programs, we must trust about 200 lines of code. This figure comes from taking a backwards slice from the statement of any individual program's correctness theorem. If we want to believe the correctness of the compiler itself, we must add additionally about 100 lines to include the formalization of the Source language as well.

7.11 Related Work

Moore produced a verified implementation of a compiler for the Piton language [63] using the Boyer-Moore theorem prover. Piton did not have the higher-order features that make my present work interesting, and proofs in the Boyer-Moore tradition have fundamentally different trustworthiness characteristics than Coq proofs, being dependent on a large reasoning engine instead of a small proof-checking kernel. However, the Piton work dealt with larger and more realistic source and target languages.

The VLISP project [39] produced a Scheme system with a rigorous but non-mechanized proof of correctness. They also made heavy use of denotational semantics, but they dealt with a dynamically-typed source language and so did not encounter many of the interesting issues reported here related to type-preserving compilation.

Semantics preservation proofs have been published before for individual phases of type-preserving compilers, including closure conversion [62]. All of these proofs that I am aware of use operational semantics, forfeiting the advantages of denotational semantics for mechanization that I have shown here.

The CompCert project [57] has used Coq to produce a certified compiler for a subset of C. Because of this source language choice, CompCert has not required reasoning about nested variable scopes, first-class functions based on closures, or dynamic allocation. On the other hand, they deal with larger and more realistic source and target languages. CompCert uses non-dependently-typed abstract syntax and operational semantics, in contrast to my use of dependent types and denotational semantics; and I focus more on proof automation.

Many additional pointers to work on compiler verification can be found in the bibliography by Dave [27].

7.12 Conclusion

I have outlined a non-trivial case study in certification of compilers for higher-order programming languages. My results lend credence to the suitability of my implementation strategy: the encoding of language syntax and static semantics using dependent types, along with the use of denotational semantics targeting a rich but formalized meta language. I have described how generic programming and proving can be used to ease the development of type-preserving compilers and their proofs, and I have demonstrated how the certification of type-preserving compilers is congenial to automated proof.

I hope to expand these techniques to larger and more realistic source and target languages. My denotational approach naturally extends to features that can be encoded directly in CIC, including (impredicative) universal types, (impredicative) existential types, lists, and trees. Handling of “effectful” features like non-termination and mutability without first compiling to first-order form (as I’ve done here in the later stages of the compiler) is an interesting open problem. I also plan to investigate further means of automating compiler correctness proofs and factorizing useful aspects of them into re-usable libraries.

There remains plenty of room to improve the developer experience in using my approach. Programming with dependent types has long been known to be tricky. I implemented my prototype by slogging through the messy details for the small number of features that I’ve included. In scaling up to realistic languages, the costs of doing so may prove prohibitive. Some recently-proposed techniques for simplifying dependently-typed Coq programming [83] may turn out to be useful.

Coercions between different dependent types appear frequently in the approach I follow. It turns out that effective reasoning about coercions in type theory requires something more than the computational equivalences that are used in systems like CIC. In Coq, this reasoning is often facilitated by adding an axiom describing the computational behavior of coercions. Ad-hoc axioms are a convenient way of extending a proof assistant’s logic, but their loose integration has drawbacks. Coq's built-in type equivalence judgment, which is applied automatically during many stages of proof checking, will not take the axioms into account. Instead, they must be applied in explicit proofs. New languages like Epigram [59] design their type equivalence judgments to facilitate reasoning about coercions. Future certified compiler projects might benefit from being developed in such environments, modulo the current immaturity of their development tools compared to what Coq offers. Alternatively, it is probably worth experimenting with the transplantation of some of these new ideas into Coq.

Chapter 8

Generic Programming and Proving

In this chapter, I will present the AutoSyntax system introduced in Section 7.6. I'll start with a more traditional generic programming perspective, building up to describing the final tool specialized to programming language formalization.

8.1 Introduction

Idiomatic functional programs use a variety of different types of tree-structured data. For instance, we commonly encounter the type of lists, where each list is either the empty list [] or a new element concatenated onto the beginning of a list with the binary :: operator. Tree-structured data is easy to analyze with recursively-defined functions, as in this definition of a function sizeL to calculate the number of "operations" required to construct a list:

$$
\begin{array}{rcl}
\mathsf{sizeL}\ [] & = & 1 \\
\mathsf{sizeL}\ (\_ :: t) & = & 1 + \mathsf{sizeL}\ t
\end{array}
$$

We can define a similar function for binary trees, built using the two *constructors* Leaf and Node.

$$\begin{aligned}\mathsf{sizeT}\ (\mathsf{Leaf}\ \_) &= 1 \\ \mathsf{sizeT}\ (\mathsf{Node}\ t_1\ t_2) &= 1 + \mathsf{sizeT}\ t_1 + \mathsf{sizeT}\ t_2\end{aligned}$$

The pattern extends to the abstract syntax trees of a small language of arithmetic expressions:

$$\begin{aligned}\mathsf{sizeE}\ (\mathsf{Const}\ \_) &= 1 \\ \mathsf{sizeE}\ (\mathsf{Neg}\ e) &= 1 + \mathsf{sizeE}\ e \\ \mathsf{sizeE}\ (\mathsf{Plus}\ e_1\ e_2) &= 1 + \mathsf{sizeE}\ e_1 + \mathsf{sizeE}\ e_2\end{aligned}$$

While the definitions of these functions are necessarily particular to the data types on which they operate, we see a common pattern among these examples. The types we are working with are all *algebraic datatypes* as supported in the popular functional programming languages Haskell and ML. There is a generic recipe for reading off one of these function definitions from the description of any algebraic datatype: for each constructor of the datatype, we include a case that adds one to the sum of the function's recursive values on the immediate arguments of that constructor.

Now imagine that we want to extend our last code example to deal with the abstract syntax of a full, industrial-strength programming language. We find ourselves stuck with the mindless work of applying the size recipe manually for every construct of the language we are implementing. Even worse, there are at least several other such *generic functions* that often appear in compilers and other language-manipulating tools.

Even outside of functional programming and programming language implementation, we find similar problems in the manipulation of structured XML data. Must we really accept the software engineering nightmare that comes from continual manual effort keeping generic functions in sync with the data types that they manipulate? Luckily, the answer is no, as the field of *generic programming* suggests many usable approaches to writing code that is polymorphic in the structure of data types.

One of the most popular of these approaches today is the *scrap your boilerplate (SYB)* family of techniques [52, 53], which has the advantage of being usable in popular Haskell compilers with no new language extensions. SYB achieves genericity through a combination of ad-hoc code generation at compile time and "unsafe" type coercions at run time. The result integrates spectacularly well with existing practical programming environments. The necessity for type coercions remains as a skeleton in the family closet whose face we never expect to see in practice.

Unfortunately, there is one sort of activity guaranteed to shine the light of day upon the dark secrets inherent in any programming technique, and that is formal verification. In mechanized programming language metatheory of the kind highlighted recently by the POPLmark Challenge [4], we set out to write proofs about programs and programming languages that are rigorous enough to convince a mechanical proof checker. What happens when we want to prove the correctness of the compiler we implement with one of today's popular generic programming approaches? Unchecked type coercions and formal mathematical modeling tend not to mix very well.

We also find ourselves looking for a new category of functionality: *generic proofs* about our generic programs. A study of the solutions submitted for the POPLmark Challenge's first benchmark problem [3] provides some statistics on the composition of formal developments. Examining the solutions to just the first step of the challenge problem, the authors find that none of the 7 submissions that use the Coq proof assistant achieves the task without proving at least 22 auxiliary lemmas. In contrast, the reference "pencil and paper" solution included with the problem description introduces only 2 such lemmas. The remaining facts are taken to be so obvious that all practicing programming language researchers will accept them without proof or even explicit statement as theorems.

While the benefits of computer formalization of proofs about programming languages are widely accepted, most people writing proofs find that today's computer theorem proving tools make it more trouble than it's worth to take this extra step. One manifestation of the difficulties is the need to prove the generic and "obvious" lemmas that we alluded to above. In this chapter, I present a tool for building some of these proofs automatically, freeing the human prover to focus on "the interesting parts."

8.1.1 Outline

In the next section, I begin by introducing a Coq idiom critical to this work: *proof by reflection*. Next, I present my technique for the case of simply-typed algebraic datatypes. With the foundation laid, I recall the setting of Section 7.6, demonstrate the difficulties with constructing support code and proofs manually, and show how to adapt my generic programming and proving approach to lifting that burden. I finish by reviewing related work and summarizing conclusions.

In the sections that follow, I will not hesitate to cut corners in the name of readability. As a result, some of the "formal" definitions that I give will be revealed as invalid under close examination. I decided that this was a better outcome than would likely result from presenting the reader with too much code in too much excruciating detail. Nonetheless, there need be no fear that the ideas are sloppy or unsound, as I have implemented them in a real system. It is available with source code and documentation at:

`http://ltamer.sourceforge.net/`

8.2 Background: Proof by Reflection

The definitional equality $\equiv$ presented in Figure 2.3 has no counterpart in the type-checking of mainstream programming languages. With the addition of inductive types in CIC, $\equiv$ grows to include rules for simplifying pattern matching and recursive function application. The CIC type-checker will now interpret quite complex programs in the course of validating larger programs that use them as types. Why is it worth designing a language with this variety of "busy types"?

We have already seen how this definitional equality can model the denotational dynamic semantics of object languages. Another compelling answer comes from the technique of proof by reflection [9]. It is a subtle approach to efficient encoding of proofs, best introduced by example. Let us consider families of proofs for this (tautological) class of formulas:

$$F \quad ::= \quad \mathsf{True} \mid F \wedge F$$

The Coq proof terms for this class are natural deduction-style proof trees that mirror the structure of the formulas themselves. Since any general sort of type inference is undecidable for CIC, the forms in which these proof trees are stored contain many "redundant" type annotations, leading to representation sizes superlinear in the sizes of the original formulas. This seems a harsh price when we could just write down an *algorithm* for generating a formula's proof from its structure. When asked to prove a formula in F, why can't we just say "run this algorithm and see for yourself"? In fact, proof by reflection allows us to do more or less that.

We want to build a trivial "decision procedure" for F. In this case, it should always answer "the formula is true," but the interesting catch is that the procedure must be *proof-producing*. To make the proof by reflection idiom work, the procedure must be *implemented in CIC*. When we use the right dependent typing, this amounts to proving the correctness of the procedure.

Following this path, we run into a block early on. In Coq, general logical formulas are represented in Prop. Prop is *open*, because new inductive definitions (modeling new logical connectives, etc.) may add new ways of building Props. Thus, CIC provides no way of pattern matching on Props. We can't write a function that deconstructs formulas programmatically, so how can our implementation possibly know which proof to build?

The trick that we use instead is the source of the description "reflection." It is related to "reflection" in popular managed object-oriented languages like Java and C#, where it is possible to obtain a runtime "data level" view of type information. In our case, we define a data structure for describing precisely the subset of Props that interests us.

```
Inductive F : Type :=
  | F_True : F
  | F_And : F → F → F
```

Along with an injection into the original domain of interest:

$$\begin{array}{rcl} \mathsf{interp} & = & \mathsf{fix}\ f\ (x : \mathsf{F}) : \mathsf{Prop.\ match}\ x\ \mathsf{with} \\ & & \quad \mid \mathsf{F\_True} \Rightarrow \mathsf{True} \\ & & \quad \mid \mathsf{F\_And}\ x_1\ x_2 \Rightarrow f(x_1) \wedge f(x_2) \end{array}$$

Now we can write a prover for this class of formulas:

$$\mathsf{prove} \quad : \quad \forall x : \mathsf{F}.\ \mathsf{interp}\ x$$

Now, for example, with

$$x = \mathsf{F\_And\ F\_True\ (F\_And\ F\_True\ F\_True)}$$

we have $\vdash \mathsf{prove}\ x : \mathsf{True} \wedge (\mathsf{True} \wedge \mathsf{True})$.

In type-checking this expression, Coq uses the definitional equality $\equiv$ to simplify $\mathsf{interp}\ x$ to $\mathsf{True} \wedge (\mathsf{True} \wedge \mathsf{True})$. We could never use the same technique to prove, for instance, falsehood, because that formula is not in the range of interp. Using the definitional equality further to simplify $\mathsf{prove}\ x$, we arrive at exactly the large proof that we would have constructed manually. The point is that the proof checker need not perform this simplification. It can content itself with verifying that this term has the proper type, not venturing into its innards.

8.3 Generic Programming and Proving for Simply-Typed Data Structures

Recall the family of examples from the introduction. I listed a number of instantiations of a hypothetical generic "size" function. For instance, for a type of trees of natural numbers, we have:

$$\begin{aligned}\mathsf{sizeT}\ (\mathsf{Leaf}\ \_) &= 1\\ \mathsf{sizeT}\ (\mathsf{Node}\ t_1\ t_2) &= 1 + \mathsf{sizeT}\ t_1 + \mathsf{sizeT}\ t_2\end{aligned}$$

We can give an inductive definition of this data type.

$$\begin{array}{l}\mathsf{Inductive\ tree : Type :=}\\ \quad \mid \mathsf{Leaf : nat} \rightarrow \mathsf{tree}\\ \quad \mid \mathsf{Node : tree} \rightarrow \mathsf{tree} \rightarrow \mathsf{tree}\end{array}$$

We want to write a truly generic size function that can be used with any simply-typed algebraic datatype. We imagine that the signature of the generic function should look something like:

$$\mathsf{size} \quad : \quad \forall T : \mathsf{inductiveType}.\ T \rightarrow \mathsf{nat}$$

Unfortunately, Coq provides no special type that categorizes inductively-defined types, and, even if it did, we would be left with the challenges of programmatic manipulation of arbitrary datatype definitions. Not surprisingly, the solution that I propose is based on the last section's subject, reflection.

8.3.1 Reflecting Constructors

What is the essence of an inductive type definition? What reflective representation should we craft for these definitions? Looking at the textual form of an inductive definition, it seems a reasonable start to say that an inductive type is a list of constructors. We can define a reflected representation of constructors like this:

$$\mathsf{con} \quad = \quad \mathsf{nat} \times \mathsf{Type}$$

The idea is that each constructor's list of arguments contains some number that are recursive, referring to the type that's being defined. The nat component of con tells how many recursive arguments the constructor has, and the Type component describes the remaining arguments. It will be a tuple type combining them into one package, or unit (the inductive type whose only value is tt) if all arguments are recursive. For the tree example, Leaf is reflected as $(0, \mathsf{nat})$, and Node as $(2, \mathsf{unit})$.

We need to make the interpretation of constructor descriptions explicit, like we did for our formula type F in the last section. The definition of the interpretation function $\mathsf{interpCon}$ relies on an auxiliary recursive function repeat that builds a tuple type by repeating a base type a specified number of times.

$$
\begin{aligned}
\mathsf{repeat} \quad &= \quad \lambda T : \mathsf{Type}.\ \mathsf{fix}\ f\ (n : \mathsf{nat}) : \mathsf{Type}.\ \mathsf{match}\ n\ \mathsf{with} \\
&\qquad\ \ \mid \mathsf{O} \Rightarrow \mathsf{unit} \\
&\qquad\ \ \mid \mathsf{S}\ n' \Rightarrow T \times f\ n' \\
\\
\mathsf{interpCon} \quad &= \quad \lambda T : \mathsf{Type}.\ \lambda c : \mathsf{con}. \\
&\qquad\ \ \mathsf{repeat}\ T\ (\pi_1\ c) \rightarrow \pi_2\ c \rightarrow T
\end{aligned}
$$

Now we're almost ready to state our initial reflective representation of entire inductive definitions. We first need to define two common inductive types. The first type family, sig, is the standard "sigma type," or "existential package," of type theory. (We met sig in Section 4.1, but I review it here.) The second type family is a generalization to existential packages whose first components are lists and whose second components are heterogeneous lists, where the type of each component is determined by applying a fixed function to the element in the corresponding position of the first list.

$\mathsf{Inductive\ sig}\ (T_1 : \mathsf{Type})\ (T_2 : T_1 \to \mathsf{Type}) : \mathsf{Type} :=$
$\mid \mathsf{ex} : \forall x : T_1.\ T_2\ x \to \mathsf{sig}\ T_1\ T_2$

$\mathsf{Inductive\ tupleF}\ (T_1 : \mathsf{Type})\ (T_2 : T_1 \to \mathsf{Type}) : \mathsf{list}\ T_1 \to \mathsf{Type} :=$
$\mid \mathsf{NilF} : \mathsf{tupleF}\ T_1\ T_2\ \mathsf{nil}$
$\mid \mathsf{ConsF} : \forall x : T_1.\ \forall \ell : \mathsf{list}\ T_1.$
$\quad T_2\ x \to \mathsf{tupleF}\ T_1\ T_2\ \ell \to \mathsf{tupleF}\ T_1\ T_2\ (\mathsf{cons}\ x\ \ell)$

In this chapter, I will abbreviate $\mathsf{sig}\ T_1\ (\lambda x : T_1.\ T_2)$ as $\Sigma x : T_1.\ T_2$, or $\Sigma x.\ T_2$ in contexts where T_1 is clear. I write $\langle x, y\rangle$ as an abbreviation for $\mathsf{ex}\ T_1\ T_2\ x\ y$ when T_1 and T_2 are clear from context. I also treat T_1 as an implicit argument of tupleF, abbreviating $\mathsf{tupleF}\ T_1\ T_2\ \ell$ as $\mathsf{tupleF}\ T_2\ \ell$.

I now define $\mathsf{ind} : \mathsf{Type} \to \mathsf{Type}$, such that $\mathsf{ind}\ T$ is the type of a reflected representation of inductively-defined T.

$$\mathsf{ind} \quad = \quad \lambda T : \mathsf{Type}.\ \Sigma \ell : \mathsf{list\ con}.\ \mathsf{tupleF}\ (\mathsf{interpCon}\ T)\ \ell$$

Here is the reflected representation rtree of tree.

$$\begin{array}{rcl} \mathsf{clist} & = & [(0, \mathsf{nat}), (2, \mathsf{unit})] \\ \mathsf{cLeaf} & = & \lambda\_ : \mathsf{unit}.\ \lambda n : \mathsf{nat}.\ \mathsf{Leaf}\ n \\ \mathsf{cNode} & = & \lambda r : \mathsf{tree} \times \mathsf{tree} \times \mathsf{unit}.\ \lambda\_ : \mathsf{unit}.\ \mathsf{Node}\ (\pi_1\ r)\ (\pi_2\ r) \\ \mathsf{rtree} & = & \langle \mathsf{clist}, \mathsf{ConsF\ cLeaf\ (ConsF\ cNode\ NilF)} \rangle \end{array}$$

8.3.2 Reflecting Recursion Principles

With the definition of ind, we can express the desired type of size formally.

$$\mathsf{size} \quad : \quad \forall T : \mathsf{Type}.\ \mathsf{ind}\ T \to T \to \mathsf{nat}$$

We're off to a good start, having replaced the informal quantification over an inductively defined type with a quantification over any type, plus a piece of "evidence" that it really behaves like an inductive type. Unfortunately, the evidence contained in an ind is not yet sufficient to let us write size. The type we've listed for size is good, but we will have to expand the definition of ind.

With an $\mathsf{ind}\ T$ in hand, we know how to *construct* values of T, but we don't yet know how to *deconstruct* values of T; that is, we aren't able to write recursive, pattern-matching functions over T. We must expand the definition of ind to require the inclusion of a *recursion principle*.

First, we define a representation for each arm of the pattern matching in a recursive definition. $\mathsf{conIH}\ T\ c$ gives the type of arms for constructor c of type T, where each is polymorphic in the range R of the function being defined.

$$\begin{array}{rcl}\mathsf{conIH} & = & \lambda T : \mathsf{Type}.\ \lambda c : \mathsf{con}.\ \forall R : \mathsf{Type}. \\ & & \quad \mathsf{repeat}\ (T \times R)\ (\pi_1\ c) \to \pi_2\ c \to R\end{array}$$

Say we're defining some recursive function f of type $T \to R$. The arm of type $\mathsf{conIH}\ T\ c$ applied to R is given two arguments: first, the recursive arguments to this particular use of constructor c, where each argument comes packaged with the result of calling f on it recursively; and second, the remaining, non-recursive arguments. Such an arm returns something in f's range.

Now we can define the representation of recursion principles, whose job it is to define a recursive function given an arm definition for each constructor of an inductive type:

$$\begin{aligned}\mathsf{rec} \;=\;& \lambda T : \mathsf{Type}.\ \lambda \ell : \mathsf{list\ con}.\\ & \quad \forall R : \mathsf{Type}.\ \mathsf{tupleF}\ (\lambda c.\ \mathsf{conIH}\ T\ c\ R)\ \ell \to (T \to R)\end{aligned}$$

We can define the recursion principle for tree, overloading the π_i notation to denote the ith projections of tupleFs, not just normal tuples.

$$\begin{aligned}\mathsf{rTree} \;=\;& \lambda R : \mathsf{Type}.\ \lambda A : \mathsf{tupleF}\ (\lambda c.\ \mathsf{conIH}\ \mathsf{tree}\ c\ R)\ \mathsf{clist}.\\ & \quad \mathsf{fix}\ f\ (x : \mathsf{tree}) : R.\ \mathsf{match}\ x\ \mathsf{with}\\ & \quad\ \ \mid \mathsf{Leaf}\ n \Rightarrow (\pi_1\ A)\ R\ \mathsf{tt}\ n\\ & \quad\ \ \mid \mathsf{Node}\ t_1\ t_2 \Rightarrow (\pi_2\ A)\ R\ ((t_1, f\ t_1), (t_2, f\ t_2), \mathsf{tt})\ \mathsf{tt}\end{aligned}$$

Now we're ready to expand the definition of ind by adding a rec component.

$$\begin{aligned}\mathsf{ind} \;=\;& \lambda T : \mathsf{Type}.\ \Sigma \ell : \mathsf{list\ con}.\\ & \quad \mathsf{tupleF}\ (\mathsf{interpCon}\ T)\ \ell \times \mathsf{rec}\ T\ \ell\end{aligned}$$

Define consOf, buildersOf, and recOf as shortcut functions for extracting the three different pieces of an ind.

We can finally define size. In this and later complicated definitions, I'll play a little fast and loose with typing, leaving out annotations that Coq really isn't able to infer. For instance, I use a function mapF for constructing a tupleF by providing a function to be applied to every element of a list; it will usually be necessary to specify the relevant dependent typing relationship (parameter T_2 of tupleF) explicitly in real code. I also rely on a function foldRepeat to duplicate the behavior of the traditional list right fold operator over tuples built with repeat, which can be thought of as lists of known length. It will also, in practice, require more explicit arguments than I will give here.

$$
\begin{array}{rcl}
\mathsf{size\ tree\ rtree} & \equiv & (\mathsf{recOf\ rtree})\ \mathsf{nat}\ (\mathsf{mapF} \\
& & \quad (\lambda c : \mathsf{con}.\ \lambda r.\ \lambda\_.\ \mathsf{foldRepeat} \\
& & \qquad (\lambda v : \mathsf{tree} \times \mathsf{nat}.\ \lambda n : \mathsf{nat}.\ \pi_2\ v + n) \\
& & \qquad 1\ r) \\
& & \quad (\mathsf{consOf\ rtree}) \\
& \equiv & (\mathsf{recOf\ rtree})\ \mathsf{nat} \\
& & \quad (\mathsf{ConsF}\ (\lambda\_.\ \lambda\_.\ 1) \\
& & \qquad (\mathsf{ConsF}\ (\lambda r.\ \lambda\_.\ \pi_2\ (\pi_1\ r) \\
& & \qquad\quad + \pi_2\ (\pi_2\ r) + 1)\ \mathsf{NilF})) \\
& \equiv & \mathsf{fix}\ f\ (x : \mathsf{tree}) : \mathsf{nat}.\ \mathsf{match}\ x\ \mathsf{with} \\
& & \quad \mid \mathsf{Leaf}\ n \Rightarrow 1 \\
& & \quad \mid \mathsf{Node}\ t_1\ t_2 \Rightarrow f\ t_1 + f\ t_2 + 1
\end{array}
$$

Figure 8.1: Simplifying one application of the generic size function

$$
\begin{array}{rcl}
\mathsf{size} & = & \lambda T : \mathsf{Type}.\ \lambda \iota : \mathsf{ind}\ T. \\
& & \quad (\mathsf{recOf}\ \iota)\ \mathsf{nat}\ (\mathsf{mapF} \\
& & \qquad (\lambda c : \mathsf{con}.\ \lambda r.\ \lambda\_.\ \mathsf{foldRepeat} \\
& & \qquad\quad (\lambda v : \mathsf{T} \times \mathsf{nat}.\ \lambda n : \mathsf{nat}.\ \pi_2\ v + n) \\
& & \qquad\quad 1\ r) \\
& & \qquad (\mathsf{consOf}\ \iota))
\end{array}
$$

Now we can examine in Figure 8.1 the simplification behavior of size applied to a revised version of the reflected representation rtree that contains the recursion principle rTree. Modulo some commutativity of addition, we arrive at exactly the definition that we started out with! The definitional equality $\equiv$ is sufficient to simplify instantiations of the generic size function into their natural forms. Thus, we can use the generic and specific versions interchangeably for type-checking (and proof-checking) purposes.

8.3.3 Generic Proofs

We could stop at this point were we only interested in solving the traditional problems of generic programming. However, the express motivation of this work is to allow the use of generic programs in concert with formal verification. It's high time that we take a look at how we may construct proofs about generic functions.

Obviously we can prove facts about *specific instantiations* of generic functions in the usual way, as the definitional equality allows us to reduce these instantiations to standard forms. However, the real promise of generic techniques in a formal theorem-proving setting is in automating *families* of proofs. We want to be able to prove that certain theorems hold for *any* instantiations of generic functions.

Unfortunately, our definition of the ind evidence packages isn't yet sufficient to allow us to prove interesting theorems. We now know how to *construct* values, and we know how to *deconstruct* them via recursive function definitions, but we don't know anything about the behavior of functions that we build in this way. The missing ingredient is a standard *fixed-point equation* decomposing applications of rec-built functions recursively.

$$\begin{array}{lll}
\mathsf{eqn} & = & \lambda T : \mathsf{Type}.\ \lambda \ell : \mathsf{list\ con}. \\
& & \quad \lambda b : \mathsf{tupleF}\ (\mathsf{interpCon}\ T)\ \ell.\ \lambda f : \mathsf{rec}\ T\ \ell. \\
& & \quad \forall R : \mathsf{Type}.\ \forall a : \mathsf{tupleF}\ (\lambda c.\ \mathsf{conIH}\ T\ c\ R)\ \ell. \\
& & \quad \forall c : \mathsf{con}.\ \forall n : \mathsf{nat}.\ \pi_n\ \ell = c\ \rightarrow \\
& & \quad \forall r : \mathsf{repeat}\ T\ (\pi_1\ c).\ \forall v : \pi_2\ c. \\
& & \quad f\ R\ a\ ((\pi_n\ b)\ r\ v) \\
& & \quad = (\pi_n\ a)\ R\ (\mathsf{mapRepeat}\ (\lambda x : T.\ (x, f\ R\ a\ x))\ r)\ v
\end{array}$$

eqn is quite a mouthful. It is defined in terms of T, the type we're manipulating; ℓ, its list of reflected constructors; b, its actual constructors; and f, its recursion principle.

We assert an equation for any range type R and any recursive function defined from T to R with pattern-matching arms a. Give the name F to the recursive function so defined. For any constructor c appearing in position n of ℓ, and any appropriate arguments r and v to the "real" version of c, we have (informally) that $F(c\ r\ v)$ is equal to the expected expansion based on the proper arm from a, which must be described in terms of recursive calls to F with the elements of r.

Now we have all we need to provide each case of an inductive proof. However, we lack the final ingredient to tie the cases together. For the sake of a concrete example, consider the silly modification of size to size2 by changing every occurrence of 1 to 2, so that instances of size2 are equivalent to doubling the results of size instances.

Like for size, for a piece of ind evidence ι, we define size2 by applying recOf ι with $R = \mathsf{nat}$. Remembering the "propositions as types" and "proofs as programs" principle, we try to construct an inductive proof that, for all x, size2 $\iota\ x$ is even. Inductive proofs are isomorphic to recursive functions, so we expect to be able to use recOf ι to build the proof.

Which value of R will let us accomplish this? Informally, we want something like $R = \mathsf{isEven}\ (\mathsf{size2}\ \iota\ x)$. Of course, this is invalid, since it contains a free variable x. What we *really* want is $R = \lambda x : T.\ \mathsf{isEven}\ (\mathsf{size2}\ \iota\ x)$. That is, R is now a *predicate* over T's, not simply a proposition; alternatively, R is an indexed family of dependent types. We must modify our definition of rec to support such families. The original formulation with non-dependent types will be derivable from this richer version.

We start by redefining conIH, the type of a single arm in a recursive definition. We add a new parameter, b, the function for applying the constructor in question. R's type is changed as described above; the type of the argument that includes recursive call results is changed to use a Σ type to reflect dependence on actual values; and the final result type uses b to build the actual value that we're talking about, so that we can apply R to it.

$$\begin{array}{lcl} \mathsf{conIH} & = & \lambda T : \mathsf{Type}.\ \lambda c : \mathsf{con}.\ \lambda b : \mathsf{interpCon}\ T\ c. \\ & & \quad \forall R : T \to \mathsf{Type}. \\ & & \quad \forall r : \mathsf{repeat}\ (\Sigma x : T.\ R\ x)\ (\pi_1\ c). \\ & & \quad \forall v : \pi_2\ c.\ R\ (b\ (\mathsf{mapRepeat} \\ & & \qquad (\lambda y : (\Sigma x : T.\ R\ x).\ \pi_1\ y)\ r)\ v) \end{array}$$

Now a simple modification to rec finishes the job. We use a type family tupleFF which is like tupleF, but is parameterized on an existing tupleF whose elements may influence the types of the corresponding elements of the new tuple.

$$\begin{array}{lcl} \mathsf{rec} & = & \lambda T : \mathsf{Type}.\ \lambda \ell : \mathsf{list\ con}.\ \lambda b : \mathsf{tupleF}\ (\mathsf{interpCon}\ T)\ \ell. \\ & & \quad \forall R : T \to \mathsf{Type}.\ \mathsf{tupleFF} \\ & & \qquad (\lambda c.\ \lambda b : \mathsf{interpCon}\ T\ c.\ \mathsf{conIH}\ T\ c\ b\ R)\ b \\ & & \qquad \to (\forall x : T.\ R\ x) \end{array}$$

We also change eqn's type to take into account rec's new type.

Though the details are tedious, it is now possible to push through a generic proof of $\forall x, \mathsf{isEven}\ (\mathsf{size2}\ \iota\ x)$. We build the proof using recOf ι with $R = \lambda x : T.\ \mathsf{isEven}\ (\mathsf{size2}\ \iota\ x)$. In each arm of the proof, corresponding to some constructor c, we start out by using eqn to rewrite the goal, revealing it (after some simplification with $\equiv$) to be a fold over the number of recursive arguments that c has. An inner induction on that number of arguments allows us to establish that the sum is even, relying crucially on the inductive hypotheses coming to us by way of the parameter r in conIH.

Though the subject of this chapter is formal theorem proving, the above barely deserves to be called a "proof sketch." However, I will stick to that level of detail for the next few portions of the chapter. Section 8.5 discusses the real engineering issues of constructing this kind of proof in Coq.

8.3.4 A Word on Trusted Code Bases

Writing programs that build programs is a tricky business. It's very easy to introduce bugs that remain hidden even after very careful testing. By implementing generic programs in Coq with rich dependent types, we get the type checker to validate their basic sanity properties. Modulo bugs in Coq, there is no chance for an unexpected input to cause a generic program to produce an ill-typed output. This is already very useful on the examples we've considered, and it will get even more useful when we move to generic programs with fancier dependent types that capture more domain-specific invariants.

Yet the reader may have noticed an obstacle in the way of achieving completely formal generic programming with my techniques. I presented no formal procedure for constructing ind evidence packages. Indeed, as for reflective proofs in general, these packages are constructed in an ad-hoc way; in this case, by OCaml code in a plug-in for Coq. This mirrors the situation for SYB and similar generic programming techniques, where some basic combinators must be constructed for each type outside of any nice static type system. Here we are slightly better off in that regard, because, once a piece of evidence is created, the code that uses it is free of the unsafe type casts that appear in the corresponding parts of SYB.

It's worth mentioning another consequence of this split for formal theorem-proving. It often happens that one wants a certified program whose specification can be given quite simply without any nasty boilerplate, but whose implementation calls out for judicious use of generic programming. My certified compiler from Chapter 7 is one such example. Though the ad-hoc generation of evidence has no proofs associated with it, *it is still not trusted* in this kind of scenario. Any evidence that pleases the type checker leads to correct code generation, and the type checker will catch faulty evidence before it can precipitate any invalid proof.

8.4 Managing de Bruijn Indices

I will present a concrete language formalization and transformation example to work with in the rest of this chapter. It follows the scheme introduced in Chapter 6, giving an even simpler simply-typed lambda calculus than before, choosing unit as the base type. The type language is:

$$\begin{array}{l}\mathsf{Inductive\ ty : Type :=}\\ \quad \mid \mathsf{Unit : ty}\\ \quad \mid \mathsf{Arrow : ty} \to \mathsf{ty} \to \mathsf{ty}\end{array}$$

The term language is:

$$\begin{array}{l}\mathsf{Inductive\ term : list\ ty} \to \mathsf{ty} \to \mathsf{Type :=}\\ \quad \mid \mathsf{Var} : \forall \Gamma : \mathsf{list\ ty}.\ \forall \tau : \mathsf{ty}.\ \mathsf{var}\ \Gamma\ \tau \to \mathsf{term}\ \Gamma\ \tau\\ \quad \mid \mathsf{UnitIntro} : \forall \Gamma : \mathsf{list\ ty}.\ \mathsf{term}\ \Gamma\ \mathsf{Unit}\\ \quad \mid \mathsf{App} : \forall \Gamma : \mathsf{list\ ty}.\ \forall \tau_1, \tau_2 : \mathsf{ty}.\ \mathsf{term}\ \Gamma\ (\mathsf{Arrow}\ \tau_1\ \tau_2)\\ \quad\quad \to \mathsf{term}\ \Gamma\ \tau_1 \to \mathsf{term}\ \Gamma\ \tau_2\\ \quad \mid \mathsf{Lam} : \forall \Gamma : \mathsf{list\ ty}.\ \forall \tau_1, \tau_2 : \mathsf{ty}.\ \mathsf{term}\ (\tau_1 :: \Gamma)\ \tau_2\\ \quad\quad \to \mathsf{term}\ \Gamma\ (\mathsf{Arrow}\ \tau_1\ \tau_2)\end{array}$$

Part of the attraction of this representation technique is that, when we write compilers and other code transformations, the Coq type system ensures that our transformations produce well-typed terms when passed well-typed terms. For a simple example, consider this function, which implements η-expansion.

$$\begin{array}{rcl} \mathsf{eta} & = & \lambda\Gamma, \tau_1, \tau_2.\ \lambda e : \mathsf{term}\ \Gamma\ (\mathsf{Arrow}\ \tau_1\ \tau_2). \\ & & \quad \mathsf{Lam}\ (\mathsf{App}\ e\ (\mathsf{Var}\ \mathsf{First})) \end{array}$$

There is a problem here. The definition of eta doesn't type-check! Where the Coq variable e is used, a term of type term $(\tau_1 :: \Gamma)$ (Arrow τ_1 τ_2) is expected, while e has type term Γ (Arrow τ_1 τ_2). We know informally that it is of course acceptable to bring an extra, unused variable (in this case, the variable bound by the new lambda) into scope, and it is exactly this fact that would convince the type-checker to accept this definition.

What we need is an auxiliary function typed like this:

$$\mathsf{lift} \quad : \quad \forall\Gamma : \mathsf{list\ ty}.\ \forall\tau, \tau' : \mathsf{ty}.\ \mathsf{term}\ \Gamma\ \tau \rightarrow \mathsf{term}\ (\tau' :: \Gamma)\ \tau$$

This is the lifting operation of de Bruijn indices. As generally happens with this kind of dependently-typed representation, lift can also be thought of as a lemma about typing derivations. In this case, it's a standard *weakening* lemma: "If $\Gamma \vdash e : \tau$, then, for x not free in e, $\Gamma, x : \tau' \vdash e : \tau$."

Now we can revise the definition of eta, such that it type-checks without incident.

$$\begin{array}{rcl} \mathsf{eta} & = & \lambda\Gamma, \tau_1, \tau_2.\ \lambda e : \mathsf{term}\ \Gamma\ (\mathsf{Arrow}\ \tau_1\ \tau_2). \\ & & \quad \mathsf{Lam}\ (\mathsf{App}\ (\mathsf{lift}\ \tau_1\ e)\ (\mathsf{Var}\ \mathsf{First})) \end{array}$$

8.4.1 Dynamic Semantics

We have this simple denotational semantics of the language:

$$
\begin{aligned}
\mathsf{tyDenote} \quad &= \quad \mathsf{fix}\ d\ (\tau : \mathsf{ty}) : \mathsf{Type} := \mathsf{match}\ \tau\ \mathsf{with} \\
&\qquad \mid \mathsf{Unit} \Rightarrow \mathsf{unit} \\
&\qquad \mid \mathsf{Arrow}\ \tau_1\ \tau_2 \Rightarrow d\ \tau_1 \rightarrow d\ \tau_2
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{termDenote} \quad &= \quad \mathsf{fix}\ d\ (\Gamma : \mathsf{list\ ty})\ (\tau : \mathsf{ty})\ (e : \mathsf{term}\ \Gamma\ \tau) \\
&\qquad\quad : \mathsf{subst\ tyDenote}\ \Gamma \rightarrow \mathsf{tyDenote}\ \tau := \\
&\qquad \mid \mathsf{Var}\ v \Rightarrow \lambda\sigma.\ \mathsf{varDenote}\ v\ \sigma \\
&\qquad \mid \mathsf{UnitIntro} \Rightarrow \lambda\_.\ \mathsf{tt} \\
&\qquad \mid \mathsf{App}\ e_1\ e_2 \Rightarrow \lambda\sigma.\ (d\ e_1\ \sigma)\ (d\ e_2\ \sigma) \\
&\qquad \mid \mathsf{Lam}\ e' \Rightarrow \lambda\sigma.\ (\lambda x.\ d\ e'\ (\mathsf{SCons}\ x\ \sigma))
\end{aligned}
$$

We can use termDenote to state the correctness property of eta-expansion:

$$
\begin{aligned}
&\forall \Gamma, \tau_1, \tau_2.\ \forall e : \mathsf{term}\ \Gamma\ (\mathsf{Arrow}\ \tau_1\ \tau_2).\ \forall \sigma : \mathsf{subst\ tyDenote}\ \Gamma. \\
&\quad \mathsf{termDenote}\ (\mathsf{eta}\ e)\ \sigma = \mathsf{termDenote}\ e\ \sigma
\end{aligned}
$$

However, what seems certain to be a trivial proof has a wrinkle in it. We need to reason about the behavior of lift to prove this theorem, which requires an induction over the structure of terms and some painful reasoning about dependent types. What we really want is a lemma like this:

$$
\begin{aligned}
\mathsf{liftSound} \quad : \quad &\forall \Gamma, \tau, \tau'.\ \forall e : \mathsf{term}\ \Gamma\ \tau. \\
&\quad \forall \sigma : \mathsf{subst\ tyDenote}\ \Gamma.\ \forall v : \mathsf{tyDenote}\ \tau'. \\
&\quad \mathsf{termDenote}\ (\mathsf{lift}\ \tau'\ e)\ (\mathsf{SCons}\ v\ \sigma) \\
&\qquad = \mathsf{termDenote}\ e\ \sigma
\end{aligned}
$$

With a tool to prove this lemma for us, the correctness of eta is established easily, with no explicit induction.

8.5 A Taste of Manual Implementation

A natural first reaction to discovering the utility of lift and liftSound is to implement them manually, specialized to this language. How hard could it be, after all? In my experience, while implementing a helper function like lift without rich types in a language like ML or Haskell can be painful, implementing it *and* its correctness lemma in Coq can be tantamount to an existential crisis. It certainly doesn't help us get to the interesting parts of a development very quickly, and such manual derivations will in fact tend to monopolize a total body of code.

To portray the difficulties therein, I will present in this section snippets of manual derivation of lift and liftSound. As a side benefit, my examples will draw in aspects of programming with general inductive types and first-class equality proofs that have rarely been included in formal publications.

8.5.1 Implementing lift

First, it quickly becomes apparent that we will need to define an auxiliary function lift′. When lifting a term, recursing inside a lambda binder adds a new type to the context, and so our workhorse function must support some additional prefix of types before the position in a context where we will insert a new type. More formally, we want the following, where $\oplus$ is list concatenation and both $\oplus$ and :: are right associative:

$$\begin{aligned}\mathsf{lift}' \quad : \quad & \forall \Gamma_1, \Gamma_2 : \mathsf{list\ ty}.\ \forall \tau, \tau' : \mathsf{ty}.\ \mathsf{term}\ (\Gamma_1 \oplus \Gamma_2)\ \tau \\ & \rightarrow \mathsf{term}\ (\Gamma_1 \oplus \tau' :: \Gamma_2)\ \tau\end{aligned}$$

Armed with our knowledge of fix expressions, we make a first attempt at coding lift'. We give ourselves a break by assuming we have an appropriately-typed library function $\mathsf{liftVar}'$ that makes the corresponding adjustment to variables. We also take good advantage of opportunities for Coq to infer some omitted arguments from the values of others.

$$\begin{array}{rcl}
\mathsf{lift}' & = & \mathsf{fix}\ f\ \Gamma_1\ \Gamma_2\ \tau\ (e : \mathsf{term}\ (\Gamma_1 \oplus \Gamma_2)\ \tau) : \forall \tau'.\ \mathsf{term}\ (\Gamma_1 \oplus \tau' :: \Gamma_2)\ \tau := \\
& & \quad \mathsf{match}\ e\ \mathsf{with} \\
& & \quad\ \ \mid \mathsf{Var}\ v \Rightarrow \lambda \tau'.\ \mathsf{Var}\ (\mathsf{liftVar}'\ v\ \tau') \\
& & \quad\ \ \mid \mathsf{UnitIntro} \Rightarrow \lambda \tau'.\ \mathsf{UnitIntro} \\
& & \quad\ \ \mid \mathsf{App}\ e_1\ e_2 \Rightarrow \lambda \tau'.\ \mathsf{App}\ (\mathsf{lift}'\ e_1\ \tau')\ (\mathsf{lift}'\ e_2\ \tau') \\
& & \quad\ \ \mid \mathsf{Lam}\ e' \Rightarrow \lambda \tau'.\ \mathsf{Lam}\ (\mathsf{lift}'\ e'\ \tau')
\end{array}$$

Here I introduce one aspect of Coq programming that I've been eliding so far. To enable decidable type-checking, Coq requires certain annotations on match expressions that deal with dependent types. In this case, we should rewrite the first line of the function body as:

$$\begin{array}{l}
\mathsf{match}\ e\ \mathsf{in}\ (\mathsf{term}\ (T_1 \oplus T_2)\ T_3) \\
\quad \mathsf{return}\ (\forall \tau'.\ \mathsf{term}\ (T_1 \oplus \tau' :: T_2)\ T_3)\ \mathsf{with}
\end{array}$$

An in clause describes the type of the expression being analyzed, providing a place to bind variables like the T_i above. A return clause describes the type of the match expression in terms of the variables bound by the in clause. In each branch arm, the T_i variables are replaced by the forms implied by the constructor being matched. For instance, in the UnitIntro branch, we have the instantiation $T_3 = \mathsf{Unit}$. This general regime should be familiar to readers accustomed to working with GADTs. In Coq, the programmer writes explicitly some of what standard GADT inference algorithms infer. In return, the Coq programmer is able to type-check more complex programs by providing explicit coercions.

Unfortunately, there is still a problem with our amended definition! Coq only allows in clauses that are inductive type names applied to lists of variables. Our code has $T_1 \oplus T_2$ where it must have a single variable. Why does Coq place this restriction? The answer is that, without the restriction, the type checker would have to solve higher-order unification problems, making type checking undecidable [48]. In other words, there can exist no algorithm that matches up variables between arbitrary in clauses and arbitrary parameter choices in the range types of constructors. This doesn't mean that there can't be useful restrictions that are decidable, or that we can't optimistically apply heuristics until some time-out is reached, but the designers of Coq chose to follow a simpler path.

A standard trick gets us out of this mess. We convert to *equality-passing style*, taking advantage of CIC's expressiveness to work with *first-class equality proofs* and their associated coercions. In Coq, equality is an inductive type like any other.

$$\begin{array}{l} \mathsf{Inductive\ eq}\ (T : \mathsf{Type})\ (x : T) : T \to \mathsf{Prop} := \\ \quad \mid \mathsf{refl\_equal} : \mathsf{eq}\ T\ x\ x \end{array}$$

We define equality (usually abbreviated with the binary = operator) as "the least reflexive relation," bootstrapping off of the definitional equality $\equiv$ built into CIC. Whereas $\equiv$ is applied implicitly, we can name equality "facts" that may or may not hold, by reifying $\equiv$ into this "propositional equality." Based on CIC's rules governing inductive types, the following operation is derivable:

$$\begin{array}{rcl} \mathsf{cast} & : & \forall T : \mathsf{Type}.\ \forall f : (T \to \mathsf{Type}).\ \forall x, y : T. \\ & & x = y \to f\ x \to f\ y \end{array}$$

It says that, for any "type with a hole in it" (represented by a function f), if we present a proof that $x = y$, and if x and y are of the right type to fill the hole, then we

$$
\begin{array}{rcl}
\mathsf{lift}'' & = & \mathsf{fix}\ f\ \Gamma\ \tau\ (e : \mathsf{term}\ \Gamma\ \tau) \\
& & \quad : \forall \Gamma_1, \Gamma_2.\ \Gamma = \Gamma_1 \oplus \Gamma_2 \\
& & \qquad \to \forall \tau'.\ \mathsf{term}\ (\Gamma_1 \oplus \tau' :: \Gamma_2)\ \tau := \\
& & \ \mathsf{match}\ e\ \mathsf{in}\ (\mathsf{term}\ T_1\ T_2) \\
& & \quad \mathsf{return}\ (\forall \Gamma_1, \Gamma_2.\ T_1 = \Gamma_1 \oplus \Gamma_2 \\
& & \qquad \to \forall \tau'.\ \mathsf{term}\ (\Gamma_1 \oplus \tau' :: \Gamma_2)\ T_2)\ \mathsf{with} \\
& & \quad \mid \mathsf{Var}\ v \Rightarrow \lambda \Gamma_1, \Gamma_2, pf, \tau'.\ \mathsf{Var}\ (\mathsf{liftVar}'' pf\ v\ \tau') \\
& & \quad \mid \mathsf{UnitIntro} \Rightarrow \lambda \Gamma_1, \Gamma_2, pf, \tau'.\ \mathsf{UnitIntro} \\
& & \quad \mid \mathsf{App}\ e_1\ e_2 \Rightarrow \lambda \Gamma_1, \Gamma_2, pf, \tau'. \\
& & \qquad \mathsf{App}\ (\mathsf{lift}''\ e_1\ pf\ \tau')\ (\mathsf{lift}''\ e_2\ pf\ \tau') \\
& & \quad \mid \mathsf{Lam}\ e' \Rightarrow \lambda \Gamma_1, \Gamma_2, pf, \tau'. \\
& & \qquad \mathsf{Lam}\ (\mathsf{lift}''\ e'\ (\mathsf{liftPf}\ pf\ \_)\ \tau')
\end{array}
$$

Figure 8.2: Correct definition of lift''

may cast type $f\ x$ to type $f\ y$. This is a computational interpretation of what it means for equality to be a congruence.

We can use equality to write the main helper function lift'' as shown in Figure 8.2. We give the function a new (but equivalent) type to the type we gave lift': it takes as input a term e that can be associated with *any* context Γ. *However*, as additional arguments, we are required to provide contexts Γ_1 and Γ_2 and a proof that $\Gamma = \Gamma_1 \oplus \Gamma_2$. Though I don't show it here, cast is used in the definition of $\mathsf{liftVar}''$. I leave underscores as "jokers" in some places where their values are inferable.

To build the proof for the recursive call in the Lam case, we rely on some implementation of this type, corresponding to an "obvious" theorem about lists and equality:

$$\mathsf{liftPf} \quad : \quad \forall \Gamma_1, \Gamma_2.\ \Gamma_1 = \Gamma_2 \to \forall \tau.\ \tau :: \Gamma_1 = \tau :: \Gamma_2$$

Now lift' is definable as

$$\lambda\Gamma_1, \Gamma_2, \tau, \tau', e.\ \mathsf{lift}''\ e\ (\mathsf{refl\_equal}\ (\Gamma_1 \oplus \Gamma_2))\ \tau'$$

8.5.2 Proving liftSound

The heart of the proof of liftSound is, probably unsurprisingly, in a lemma about lift$''$. I overload :: and $\oplus$ to denote one- and multi-element concatenation of substitutions.

$$\begin{aligned}\mathsf{liftSound}'' \quad : \quad & \forall\Gamma, \tau.\ \forall e : \mathsf{term}\ \Gamma\ \tau.\ \forall\Gamma_1, \Gamma_2. \\ & \forall pf : (\Gamma = \Gamma_1 \oplus \Gamma_2).\ \forall\tau', \sigma_1, v, \sigma_2. \\ & \quad \mathsf{termDenote}\ (\mathsf{lift}''\ e\ pf\ \tau')\ (\sigma_1 \oplus v :: \sigma_2) \\ & \quad = \mathsf{termDenote}\ (\mathsf{cast}\ (\lambda X.\ \mathsf{term}\ X\ \_)\ pf\ e)\ (\sigma_1 \oplus \sigma_2)\end{aligned}$$

The most interesting part is the use of cast. Without it, we would have an argument type incompatibility for the second use of termDenote, since e is in context Γ and $\sigma_1 \oplus \sigma_2$ is in context $\Gamma_1 \oplus \Gamma_2$. By presenting an explicit equality proof between those two types, we bridge the gap.

The skeleton of the proof is an induction on the structure of e. Following the "proofs as programs" principle, this can be written as a fix expression over e. In this case, it's more convenient to use Coq's tactic-based theorem proving mode to construct the proof term interactively, with help from decision procedures. I will show only the simplest case of the proof, that for UnitIntro, as it already demonstrates how slippery this kind of proof is.

The initial proof state appears in Figure 8.3. We have a set of hypotheses/free variables appearing above the horizontal line with their types. Below the line appears the proposition we are trying to prove (the "conclusion"); alternatively, what we see there is a

$$\frac{\begin{array}{lll} \ldots & & \\ pf & : & \Gamma = \Gamma_1 \oplus \Gamma_2 \\ \ldots & & \end{array}}{\begin{array}{l} \mathsf{termDenote}\ (\mathsf{lift}''\ \mathsf{UnitIntro}\ pf\ \tau')\ (\sigma_1 \oplus v :: \sigma_2) \\ \quad = \mathsf{termDenote}\ (\mathsf{cast}\ (\lambda X.\ \mathsf{term}\ X\ \_) \\ \qquad pf\ \mathsf{UnitIntro})\ (\sigma_1 \oplus \sigma_2) \end{array}}$$

Figure 8.3: Partial initial proof state for UnitIntro case of liftSound″

type, and we are trying to code a program with that type. We can simplify the lefthand side of the conclusion to tt using only the definitional equality $\equiv$. The tricky part is simplifying the righthand side to the same value.

We might think we ought to be able to replace the cast term with UnitIntro. However, here UnitIntro has an elided parameter specifying which typing context it lives in, Γ. So, erasing the cast, we are left with the argument type incompatibility that we added the cast to avoid. A somewhat counterintuitive trick helps us make the next step. We use pf to replace Γ with $\Gamma_1 \oplus \Gamma_2$ everywhere that Γ appears, *even in* pf*'s own type*! Now pf has type $\Gamma_1 \oplus \Gamma_2 = \Gamma_1 \oplus \Gamma_2$, and we have made some good progress. Now we would be left with a *well-typed* righthand side if we made the simplification we want to make; what remains is to figure out how to justify the rewrite.

Surprisingly enough, working from just the simple inductive definition of eq that we gave earlier, we can't make further progress. CIC just isn't a strong enough type theory to model the computational behavior of casting without the addition of further rules. To fix this, it's common to add some *axiom* characterizing this behavior. Axioms are propositions asserted without proof. They are convenient ways to make global changes to the logic one is working in, though it is always critical to check that your set of axioms introduces no inconsistency. The axiom that I chose to depend on [84] is included in the Coq standard library:

$$\begin{aligned}\mathsf{castEq} \quad : \quad & \forall T : \mathsf{Type}.\ \forall f : (T \to \mathsf{Type}).\ \forall x : T. \\ & \forall pf : x = x.\ \forall v : f\ x.\ \mathsf{cast}\ f\ pf\ v = v\end{aligned}$$

One rewrite with castEq, followed by simplification using $\equiv$ alone, reduces the conclusion to $\mathsf{tt} = \mathsf{tt}$, which is proved directly by refl_equal.

8.5.3 The Prognosis

It's possible, with enough patience, to craft these programs and proofs manually, but it remains an intellectually challenging problem in each iteration. Researchers who don't specialize in type theory but want to formalize their programming languages would be justified in recoiling at the thought of needing to learn the type theory arcana I've employed.

We would rather have all of this done automatically. The more of code and proof generation we can have validated statically, the better. Though this will require even more wizardry in the implementation of the generic programming system, it will be worth the effort, because users will be able to employ the system without understanding those internals, while retaining the benefits of their rigorous verification.

8.6 The Lambda Tamer AutoSyntax System

The AutoSyntax piece of the Lambda Tamer system uses the techniques I've presented to provide generic implementations of functions like lift and lemmas like $\mathsf{liftSound}$. To do so, it uses reflected representations of inductive definitions of term languages, much as we did for simply-typed datatypes in Section 8.3. However, our situation is now more complicated, as inductive definitions like term's from the last section use universal quantification and manipulate typing contexts.

Luckily, a very regular structure suffices for the types of AST constructors. Each type begins with a $\forall$ quantification over a typing context Γ. We then have some quantifications over types and other data (such as an integer as an argument to an integer constant constructor). In the scope of these quantifiers are some variable and term arguments. Each variable has context Γ, and each has a type expressed as a function of the quantified variables. Recursive term instances are a bit more complicated. Each has a type expressed in the same way, but we also need to allow for new types to be "pushed onto the front of" Γ. Thus, we allow the context argument of a subterm to be any number of types consed onto the beginning of Γ. Finally, the result type of any constructor is the term type itself at context Γ and some type (written, like before, as a function of the quantified variables).

With this pattern in mind, we can redo the definition of con from Section 8.3:

$$\begin{aligned}\mathsf{con} \quad = \quad & \lambda ty : \mathsf{Type}.\ \Sigma T : \mathsf{Type}.\ \mathsf{list}\ (T \to ty) \\ & \times \mathsf{list}\ (T \to \mathsf{list}\ ty \times ty) \\ & \times (T \to ty)\end{aligned}$$

con is parameterized by the type ty of object language types. Each con contains a type T that combines the domains of all the "real" constructor's quantified variables, using tupling if necessary; a list of the types of variables; a list of the types of subterms; and the result type. For later convenience, we define projection functions for the four components of cons: $\mathsf{quantsOf}$, varsOf, $\mathsf{termsOf}$, and $\mathsf{resultOf}$.

A few examples should serve to explain the pattern. A good $\mathsf{con\ ty}$ for the constructor $\mathsf{UnitIntro}$ of our running example is:

$$\langle \mathsf{unit}, (\mathsf{nil}, \mathsf{nil}, \lambda\_.\ \mathsf{Unit})\rangle$$

The case of Var illustrates variable typing:

$$\langle \mathsf{ty}, ([\lambda\tau.\ \tau], \mathsf{nil}, \lambda\tau.\ \tau)\rangle$$

Finally, the case of Lam illustrates binder typing:

$$\langle \mathsf{ty} \times \mathsf{ty}, (\mathsf{nil}, [\lambda t.\ ([\pi_1\ t], \pi_2\ t)], \lambda t.\ \mathsf{Arrow}\ (\pi_1\ t)\ (\pi_2\ t))\rangle$$

From this starting point, we can redefine $\mathsf{interpCon}$, conIH, rec, eqn, and ind in the (more or less) obvious ways. Since the new definitions are messier but unsurprising, I won't include them here. (The interested reader can, of course, obtain them in the source distribution.) The AutoSyntax plug-in for Coq generates all of the corresponding pieces automatically by querying the Coq environment for the definitions of types that the user requests for code generation.

8.6.1 Restricting Denotations

AutoSyntax also contains generic proofs of lemmas like $\mathsf{liftSound}$. Looking back at that theorem's statement, we see that it depends in no way on the details of the language under analysis. Its statement follows a pattern that recurs in our setting, where we give programs "denotational semantics" via definitional compilers. We want to know that a particular "commutative diagram" holds, in a sense specific to the generic function in question. We want to know that we get the same result by "compiling" expression e directly with substitution σ as we get by applying the generic function to e, applying some compensating transformation to σ, and *then* compiling.

How can we write a proof of this theorem that works for *any* definitional compiler? Actually, it's clear that we can't make it work for just *any* dynamic semantics, since it's easy to come up with perverse semantics that perform intensional analysis on terms. For instance, we could write a denotation function that checks if its input equals a particular term and then returns some arbitrary result; otherwise, it uses a sane recursive definition. The $\mathsf{liftSound}$ specification might not hold on the term that's been singled out. Now the question is what conditions we can impose on denotations to facilitate use of a generic proof technique without compromising flexibility in language definition.

We find our criterion in the standard sanity check of denotational semantics, *compositionality.* Roughly speaking, we want the meaning of a term to be a function of nothing but the constructor used to build it, the values given to its quantified variables, and *the denotations of* its variables and subterms. In particular, it shouldn't be valid to examine the *syntax* of a subterm in a denotation function.

For our specific example language, this means that we want there to exist functions

$$
\begin{array}{rcl}
f_{\mathsf{Var}} & : & \forall \tau.\ \mathsf{tyDenote}\ \tau \to \mathsf{tyDenote}\ \tau \\
f_{\mathsf{UnitIntro}} & : & \mathsf{tyDenote}\ \mathsf{Unit} \\
f_{\mathsf{App}} & : & \forall \tau_1, \tau_2.\ \mathsf{tyDenote}\ (\mathsf{Arrow}\ \tau_1\ \tau_2) \\
 & & \quad \to \mathsf{tyDenote}\ \tau_1 \to \mathsf{tyDenote}\ \tau_2 \\
f_{\mathsf{Lam}} & : & \forall \tau_1, \tau_2.\ (\mathsf{tyDenote}\ \tau_1 \to \mathsf{tyDenote}\ \tau_2) \\
 & & \quad \to \mathsf{tyDenote}\ (\mathsf{Arrow}\ \tau_1\ \tau_2)
\end{array}
$$

Notice that no detail of the implementation of $\mathsf{tyDenote}$ comes into play; we hope that it's initially plausible that function signatures like these can be read off from the definition of a term type. It's just a "coincidence" of the closeness of our object language to CIC that f_{Var}, f_{App}, and f_{Lam} ending up having the types of particular identity functions.

A denotational semantics for our object language is *compositional* when there exist functions of these types that satisfy the following equations. We overload juxtaposition as notation for looking up a variable in a substitution.

$$
\begin{array}{rcl}
\mathsf{termDenote}\ (\mathsf{Var}\ v)\ \sigma & = & f_{\mathsf{Var}}\ (\sigma v) \\
\mathsf{termDenote}\ \mathsf{UnitIntro}\ \sigma & = & f_{\mathsf{UnitIntro}} \\
\mathsf{termDenote}\ (\mathsf{App}\ e_1\ e_2)\ \sigma & = & f_{\mathsf{App}}\ (\mathsf{termDenote}\ e_1\ \sigma)\ (\mathsf{termDenote}\ e_2\ \sigma) \\
\mathsf{termDenote}\ (\mathsf{Lam}\ e')\ \sigma & = & f_{\mathsf{Lam}}\ (\lambda x.\ \mathsf{termDenote}\ e'\ (\mathsf{SCons}\ x\ \sigma))
\end{array}
$$

8.6.2 Reflecting Denotations

We need to formalize the requirements of compositionality, so that we can write generic proofs about arbitrary compositional denotation functions. To start with, here is a function that calculates the proper type for one of the $f_{\_}$'s above, given a constructor and a type denotation function:

$$\begin{array}{rcl}\mathsf{schema} & = & \lambda ty : \mathsf{Type}.\ \lambda c : \mathsf{con}\ ty.\ \lambda d : (ty \to \mathsf{Type}). \\ & & \quad \forall q : \mathsf{quantsOf}\ c.\ \mathsf{tupleF}\ (\lambda v.\ d\ (v\ q))\ (\mathsf{varsOf}\ c) \\ & & \quad\ \to \mathsf{tupleF}\ (\lambda t.\ \mathsf{tupleF}\ d\ (\pi_1\ (t\ q)) \\ & & \quad\quad \to d\ (\pi_2\ (t\ q)))\ (\mathsf{termsOf}\ c) \\ & & \quad\ \to d\ ((\mathsf{resultOf}\ c)\ q)\end{array}$$

Now we can define what it means for a denotation function to be compositional with respect to a single constructor. I appeal to the reader's intuition for the new types associated with functions like interpCon.

$$\begin{array}{rcl}\mathsf{isComp} & = & \lambda ty : \mathsf{Type}.\ \lambda T : \mathsf{list}\ ty \to ty \to \mathsf{Type}. \\ & & \quad \lambda c : \mathsf{con}\ ty.\ \lambda b : \mathsf{interpCon}\ T\ c. \\ & & \quad \lambda dt : (ty \to \mathsf{Type}). \\ & & \quad \lambda dT : (\forall \Gamma, \tau.\ T\ \Gamma\ \tau \to \mathsf{subst}\ dt\ \Gamma \to dt\ \tau). \\ & & \quad \Sigma f : \mathsf{schema}\ ty\ c\ dt.\ \forall qs, vs, es, \sigma. \\ & & \quad\ dT\ (b\ qs\ vs\ es)\ \sigma \\ & & \quad\ = f\ qs\ (\mathsf{mapF}\ (\lambda v.\ \sigma v)\ vs) \\ & & \quad\quad (\mathsf{mapF}\ (\lambda e.\ \lambda vs.\ dT\ (\pi_2\ e)\ (vs \oplus \sigma))\ es)\end{array}$$

With this definition, it's easy to define full compositionality by asserting that isComp must hold for each constructor of a type.

8.6.3 Sketch of a Generic liftSound$''$ Proof

I wrap up the discussion of implementation techniques with a quick sketch of the proof of liftSound$''$ for an AST type T reflected into evidence package ι, with respect to denotation function dT that has been shown compositional. I won't provide a detailed definition of the generic lift$''$ itself. Its basic idea is straightforward: recurse through all term structure using recOf ι, applying liftVar$''$ to all variables encountered.

1. The proof is by induction on the structure of the term e. Concretely, we apply the "induction principle" found in $\mathsf{recOf}\ \iota$. Perform the following steps for each inductive case, where each corresponds to some constructor c that was used to build e.

2. Just as in Section 8.5.2, use the proof $pf : \Gamma = \Gamma_1 \oplus \Gamma_2$ to rewrite Γ to $\Gamma_1 \oplus \Gamma_2$ everywhere, and then use castEq to remove the use of cast.

3. Simplify each side of the equality by rewriting with $\mathsf{eqnOf}\ \iota$. That is, we unfold the definition of lift'' one level, as we know the top level structure of its arguments.

4. Use the compositionality of dT to rewrite the $dT\ (c\ \cdots)\ \_$ on each side of the equation with f_c. Now each side is f_c applied to the same quantified variable values, followed by different lists of denotations of object-language variables and subterms.

5. Rewrite each lefthand side variable denotation to match its righthand side counterpart using $\mathsf{liftVar''\_sound}$, a lemma from the Lambda Tamer library. (This is an inner induction on the number of variables.)

6. Rewrite each lefthand side subterm denotation to match its righthand side counterpart using the inductive hypothesis supplied by $\mathsf{recOf}\ \iota$. (This is an inner induction on the number of subterms.)

7. The conclusion is reduced to a trivial equality, which we prove by reflexivity.

8.6.4 Mutually-Inductive AST Types

Mutually-recursive type definitions are common in programming languages. Figure 7.15 provided an example, in the form of the term languages for Linear. It turns out that

Coq is expressive enough to let us build support for mutually-inductive types on top of what I've already described. I illustrate this by the simple example of two mutually inductive term types $\mathsf{term1}$ and $\mathsf{term2}$ associated with a type language ty. First, we define a type-carrying enumeration of the different term sorts:

$$\begin{array}{l} \mathsf{Inductive\ which : Type} := \\ \quad \mid \mathsf{Term1 : ty} \to \mathsf{which} \\ \quad \mid \mathsf{Term2 : ty} \to \mathsf{which} \end{array}$$

The actual AutoSyntax implementation allows the choice of distinct type languages for typing contexts versus term types. We can take advantage of this by using which as our new term type language. Now we can define a new, combined term type:

$$\begin{array}{rcl} \mathsf{term} & = & \lambda\Gamma : \mathsf{list\ ty}.\ \lambda x : \mathsf{which}.\ \mathsf{match}\ x\ \mathsf{with} \\ & & \quad \mid \mathsf{Term1}\ \tau \Rightarrow \mathsf{term1}\ \Gamma\ \tau \\ & & \quad \mid \mathsf{Term2}\ \tau \Rightarrow \mathsf{term2}\ \Gamma\ \tau \end{array}$$

It's a straightforward exercise in type-level computation to build an ind package that treats $\mathsf{term1}$ and $\mathsf{term2}$ like two parts of a single inductive type term. The AutoSyntax Coq plug-in does this work for arbitrary mutual definitions, and it translates the results back to the mutual setting.

8.6.5 Evaluation

My most extensive case study for AutoSyntax to date is on the certified compiler from Chapter 7. There, AutoSyntax is applied to the statically-typed target languages of CPS conversion, closure conversion, and conversion to closed first-order programs with explicit allocation. The portion of the source code responsible for these pieces weighs in at under 3000 lines. This includes the definitions of the syntax, static semantics, and dynamic semantics of all languages; the definitions of the transformations, which are dependently-typed in a way that brings type preservation theorems "for free"; and the semantics preservation proofs, which use Coq's tactic-based proof search facility. Among the generic functions provided by AutoSyntax are lifting, permutation, free variable set calculation, and strengthening (removing unused variables from a typing context, critical to efficient closure construction), along with derived versions of these operations, such as lifting multiple unused variables into a context at once. Following standard idioms for writing the generic pieces manually, the amount of code would have more than doubled.

I also put together a stand-alone case study of certified CPS conversion for simply-typed lambda calculus, found in `examples/CPS.v` in the source distribution. It contains about 250 lines of code. Notably, its proofs achieve almost the level of conciseness of their usual pencil-and-paper equivalents. Hypotheses in specific proof contexts are never mentioned by name. Rather, the lemmas generated by AutoSyntax are sufficient to guide relatively simple proof automation, augmented by a few "hints" about domain-specific approaches to getting out of tricky situations. I think this makes my implementation unique among all that I am aware of, with all competitors relying on considerable manual proving effort.

8.7 Related Work

In dynamically-typed languages like those in the Lisp family, simple reflective capabilities make it clear how to write generic functions, but the lack of static validation can make it difficult to get their implementations right. Approaches exist [54] for statically-typed languages that convert to untyped form and back around generic function invocations, but they retain most of the same deficiency.

Several language designs exist that embody *polytypic programming* [49, 44], where functions can be defined over the structure of types. None of these systems permit functions generic in definitions of GADTs, let alone the dependent inductive types I use in AutoSyntax. They also provide no support for theorem proving.

Derivable type classes [45] is a polytypic programming extension to Haskell, designed to tackle the practical issues of extending a production language. The same limitations apply as in the last paragraph.

Recent tools designed to work with out-of-the-box Haskell (with GHC extensions) include Scrap Your Boilerplate [52, 53] and RepLib [89]. RepLib in particular uses reflected type representations very similar to mine. Both approaches benefit from easy integration with a standard programming language, but, because of Haskell's relatively weak type system, they must resort to unsafe type casts in their implementations. Both support mixing generic function implementations that are read off directly from type structure with custom implementations for particular types. Again, neither handles generic proving. Even how to describe their implementations in a form that could be embedded in a type theory-based theorem prover like Coq is unclear; they are given using language constructs that permit non-termination, which is incompatible with type theoretical consistency.

Altenkirch and McBride [1] consider an approach reminiscent of what we presented in Section 8.3 (minus the generic proving) for the OLEG proof assistant. Pfeifer and Rueß [74] and Benke et al. [6] go further and add (in different ways) rudimentary generic proving to the same general program for Lego and Agda, respectively. The first of these two works does not treat dependent inductive types (such as I use for representing ASTs), and neither work presents generic proofs for such types, nor shows how to apply these techniques to simplifying programming language metatheory or certification of code transformations.

AutoSyntax is specialized to my language representation approach, which fixes the de Bruijn convention for variables. It could be adapted to alternate first-order choices, including those based on nominal logic [87]. Higher-order abstract syntax (HOAS) [75] is a completely different variable representation technique that uses the variables of the meta language to stand for the variables of the object language. Thus, notions like substitution and weakening are inherited "for free" from the meta language. HOAS is incompatible with CIC, but some generalization of my approach may still be possible to facilitate different kinds of generic programming and proving. I note that it may be worth the pain of explicit variable management to retain Coq's support for statically-validated manipulation of first-class meta-proofs.

8.8 Conclusion

I have presented a new approach to statically-validated generic programming and proving for classes of inductive types. The approach is compatible with small-trusted-code-base formal theorem proving, and my particular AutoSyntax system eases significantly the development of formal programming language metatheory and certified code transformations. As Section 8.6 demonstrates, the technique must be instantiated manually to suit different domain-specific uses of dependent typing. I believe it likely that a broad range of domains stand to benefit from the construction of such instantiations.

Part III

Conclusion

In this thesis, I have supported the assertion that theorem-proving tools based on *type theory* have several important and unique advantages for the practical certification of software. Classical verification approaches which separate programming from proving have failed to catch on in the real world. Type systems, on the other hand, have been widely adopted. By doing certified programming in a type theoretical setting, we capture some of the primary advantages of type systems while arriving at the same total correctness guarantees that classical verification promised.

There are a wide variety of tools for mechanical theorem proving, but only those based on type theory, such as Coq, share these advantages, each of which I've demonstrated in the preceding case studies:

- Type theory is unmatched for providing a tiny foundation for mathematics. A small type checker for a dependently-typed lambda calculus suffices for validating the full range of mathematical proofs and formal verifications.

- Types provide a natural paradigm for including specifications in all parts of a program. Where the ESC approach to verification [35] uses several ad-hoc annotation directives to label program pieces with specifications, *dependent typing* provides one simple metaphor that can be used to annotate any piece with its expected behavior.

- Programming in type theory facilitates verifications that depend on solving undecidable or intractable problems, while keeping type checking and proof checking decidable. Via the Curry-Howard correspondence, it is possible to *program with proofs* in a normal functional programming style, allowing the programmer to provide correctness justifications for his program inline with the source code. *Program extraction* mechanisms can produce executable programs by erasing these proofs, removing any runtime efficiency cost that they may have added.

- At the same time, classical methods from *tactic-based theorem proving* can be brought to bear in this setting, with some perhaps unforeseen benefits. The same tactic mechanism can be thought of as a theorem-proving tool or as *a method of writing dependently-typed programs*. We can view theorem proving ideas as programming ideas, as in applying "proof search" to synthesize programs from their specifications; and we can view programming ideas as theorem proving ideas, as in writing programs that synthesize proofs from inputs.

- The idea of *higher-order combinators* from functional programming adapts easily to certified programming in type theory. It is possible to write combinators that not only combine programs but also *combine their specifications and correctness proofs*, without requiring that any new proofs be written upon use. These combinators can be polymorphic in specifications and even in types that may be richly specified or may not be.

- ML-style module systems adapt readily to dependent type theory, providing a unified description of how associations of programs and proofs can be composed while maintaining appropriate encapsulation.

- General inductive types allow great flexibility in defining richly-specified types without ever writing anything that looks like a logical specification, following the trend embodied in generalized algebraic datatypes. For instance, it is straightforward to define abstract syntax tree types that enforce object language typing rules. Transformations on those trees can be validated as *type-preserving* by simple type-checking, without the need to write any "proofs" of the traditional kind.

- By working in a type theory with a rich *definitional equality*, it is possible to model the behavior of many programming languages by compiling them into the underlying logic, with no need to write an explicit dynamic semantics.

- The *proof by reflection* technique also takes advantage of the definitional equality to allow us to write *certified decision procedures*. Classes of logical goals can be specified with abstract syntax trees. By writing translations from trees into formulas, we are able to give static types to procedures that produce families of proofs. Proof by reflection enables a very effective kind of *statically-validated metaprogramming*, whether for "programs" or for "proofs."

Much work remains to be done in simplifying both the "programming" mode, through novel language features; and the "proving" mode, through effective and customizable automation support. Nonetheless, I believe that, in the last few years, we have reached a "sweet spot" for certified programming. Hardware has reached levels of performance that make aggressive proof automation possible, and the engineering effort put into proof assistants like Coq has reached a stage where they are truly useful tools, not just testbeds for research on theorem proving. I hope that the examples of certified programming idioms that I have demonstrated in this dissertation can help provide inspiration for tools that will one day be part of every programmer's arsenal.

Bibliography

[1] Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Proc. the IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 1–20, 2003.

[2] Andrew W. Appel. Foundational proof-carrying code. In *Proc. the 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, pages 247–256, June 2001.

[3] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, and Stephanie Weirich. Engineering aspects of formal metatheory, April 2007. Submitted for publication.

[4] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In *Proc. the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'05)*, pages 50–65, August 2005.

[5] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. the 8th International SPIN Workshop on Model Checking of Software (SPIN'01)*, pages 103–122, May 2001.

[6] Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic J. of Computing*, 10(4):265–289, 2003.

[7] Yves Bertot. Formalizing a JVML verifier for initialization in a theorem prover. In *Proc. the 13th International Conference on Computer Aided Verification (CAV'01)*, pages 14–24, July 2001.

[8] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[9] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *Proc. the 14th Symposium on Theoretical Aspects of Computer Software (STACS'97)*, pages 515–529, February 1997.

[10] Robert S. Boyer and J. Strother Moore. Proving theorems about LISP functions. *J. ACM*, 22(1):129–144, 1975.

[11] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. In *Proc. the Fifth Annual IEEE Symposium on Logic in Computer Science (LICS'90)*, pages 428–439, June 1990.

[12] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1):56–78, September 2005.

[13] Bor-Yuh Evan Chang, Adam Chlipala, and George C. Necula. A framework for certified program analysis and its applications to mobile-code safety. In *Proc. the Seventh*

International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06), pages 174–189, January 2006.

[14] Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. The Open Verifier framework for foundational verifiers. In *Proc. the 2nd ACM Workshop on Types in Language Design and Implementation (TLDI'05)*, pages 1–12, January 2005.

[15] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *Proc. the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, pages 66–77, September 2005.

[16] Adam Chlipala. Modular development of certified program verifiers with a proof assistant. In *Proc. the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*, pages 160–171, September 2006.

[17] Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proc. the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, pages 54–65, June 2007.

[18] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, May 1981.

[19] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving that programs eventually do something good. In *Proc. the 34th An-*

nual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07), pages 265–276, January 2007.

[20] C. Coquand and T. Coquand. Structured type theory. In *Proc. the First Workshop on Logical Frameworks and Meta-languages (LFM 1999)*, July 1999.

[21] Thiery Coquand and Gerard Huet. Constructions: A higher order proof system for mechanizing mathematics. In *Proc. European Conf. on Computer Algebra (EUROCAL 1985)*, pages 151–184, April 1985.

[22] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference.* 2006.

[23] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. the 4th ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 234–252, January 1977.

[24] Karl Crary. Programming language semantics in foundational type theory. In *Proc. the IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods (PROCOMET'98)*, pages 107–125, June 1998.

[25] Karl Crary. Toward a foundational typed assembly language. In *Proc. the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 198–212, January 2003.

[26] D. Dams and K. Namjoshi. Shape analysis through predicate abstraction and model

checking. In *Proc. the Fourth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'03)*, pages 310–324, January 2003.

[27] Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.

[28] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formal manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[29] David Delahaye. A tactic language for the system Coq. In *Proc. the 7th International Conference on Logic for Programming and Automated Reasoning (LPAR'00)*, pages 85–95, November 2000.

[30] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[31] The Coq development team. The Coq proof assistant user's guide. Version 8.1. Technical report, INRIA, 2006.

[32] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[33] Pollack *et.al.* The LEGO Proof Assistant, 2001.

[34] Xinyu Feng, Zhaozhong Ni, Zhong Shao, and Yu Guo. An open framework for foundational proof-carrying code. In *Proc. the 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*, pages 67–78, January 2007.

[35] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, pages 234–245, June 2002.

[36] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[37] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In *Selected papers from the International Workshop on Types for Proofs and Programs (TYPES'94)*, pages 39–59, 1995.

[38] Michael Gordon. HOL: A machine oriented formulation of higher-order logic. Technical Report 85, University of Cambridge, Computer Laboratory, July 1985.

[39] Joshua D. Guttman, John D. Ramsdell, and Mitchell Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995.

[40] Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. *J. Automated Reasoning*, 31(3-4):191–229, 2003.

[41] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. of the ACM*, 40(1):143–184, January 1993.

[42] T. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction

refinement. In *Proc. the Fifteenth International Conference on Computer-Aided Verification (CAV'03)*, pages 262–274, July 2003.

[43] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proc. the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, pages 58–70, January 2002.

[44] Ralf Hinze. A new approach to generic functional programming. In *Proc. the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*, pages 119–132, January 2000.

[45] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In *Proc. the ACM SIGPLAN Haskell Workshop*, September 2000.

[46] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:567–580, 1969.

[47] Furio Honsell and Donald Sannella. Pre-logical relations. In *Proc. the 13th International Computer Science Logic Workshop (CSL'99)*, pages 546–561, September 1999.

[48] G. Huet. The undecidability of unification in third order logic. *Information and Control*, 22:257–267, 1973.

[49] Patrik Jansson and Johan Jeuring. PolyP – a polytypic programming language extension. In *Proc. the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 470–482, January 1997.

[50] Matt Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Trans. Softw. Eng.*, 23(4):203–213, 1997.

[51] Gerwin Klein and Tobias Nipkow. Verified lightweight bytecode verification. *Concurrency – Practice and Experience*, 13(1):1133–1151, 2001.

[52] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *Proc. the 1st ACM Workshop on Types in Language Design and Implementation (TLDI'03)*, pages 26–37, January 2003.

[53] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proc. the Ninth ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, pages 244–255, September 2004.

[54] Ralf Lämmel and Joost Visser. Typed combinators for generic traversal. In *Proc. the 4th International Symposium on Practical Aspects of Declarative Languages (PADL'02)*, pages 137–154, January 2002.

[55] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *Proc. the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 173–184, January 2007.

[56] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proc. the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, pages 364–377, January 2005.

[57] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*, pages 42–54, January 2006.

[58] David MacQueen. Modules for Standard ML. In *Proc. the 1984 ACM Symposium on LISP and Functional Programming (LFP'84)*, pages 198–207, August 1984.

[59] Conor McBride and James McKinna. The view from the left. *J. Functional Programming*, 14(1):69–111, 2004.

[60] Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. In *Proc. the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, pages 468–479, June 2007.

[61] Robin Milner. Implementation and applications of Scott's logic for computable functions. In *Proc. the ACM Conference on Proving Assertions about Programs*, pages 1–6, 1972.

[62] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. Technical Report CMU-CS-FOX-95-05, Carnegie Mellon University, July 1995.

[63] J. Strother Moore. A mechanically verified language implementation. *J. Automated Reasoning*, 5(4):461–492, 1989.

[64] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic

typed assembly language. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, May 1999.

[65] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *J. Functional Programming*, 13(5):957–959, 2003.

[66] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. on Programming Languages and Systems*, 21(3):527–568, 1999.

[67] George C. Necula. Proof-carrying code. In *Proc. the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 106–119, January 1997.

[68] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proc. the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*, pages 333–344, June 1998.

[69] Greg Nelson and Derek Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming Languages and Systems*, 1(2):245–257, October 1979.

[70] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proc. the 11th International Conference on Automated Deduction (CADE-11)*, pages 748–752, June 1992.

[71] Emir Pasalic, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *Proc. the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, pages 218–229, October 2002.

[72] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In *Proc. the International Conference on Typed Lambda Calculi and Applications (TLCA'93)*, pages 328–345. March 1993.

[73] L. C. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828:xvii + 321, 1994.

[74] Holger Pfeifer and Harald Rueß. Polytypic proof construction. In *Proc. the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99)*, pages 55–72, September 1999.

[75] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proc. the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*, pages 199–208, June 1988.

[76] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *Proc. the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, 1999.

[77] Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

[78] G. D. Plotkin. Lambda-definability and logical relations. Memorandum SAI-RM-4, University of Edinburgh, 1973.

[79] Gordon D. Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[80] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. the 5th International Symposium on Programming*, pages 337–350, 1982.

[81] Tim Sheard. Languages of the future. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*, pages 116–119, October 2004.

[82] Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In *Proc. the 9th European Symposium on Programming Languages and Systems (ESOP'00)*, pages 366–381, March 2000.

[83] Matthieu Sozeau. Subset coercions in Coq. In *Selected papers from the International Workshop on Types for Proofs and Programs (TYPES'06)*, 2006.

[84] T. Streicher. Semantical investigations into intensional type theory. Habilitationsschrift, LMU München, 1993.

[85] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: a type-directed optimizing compiler for ML. In *Proc. the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI '96)*, pages 181–192, May 1996.

[86] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.

[87] C. Urban and C. Tasson. Nominal techniques in Isabelle/HOL. In *Proc. the 20th International Conference on Automated Deduction (CADE-20)*, pages 38–53, July 2005.

[88] Philip Wadler. The essence of functional programming. In *Proc. the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92)*, pages 1–14, January 1992.

[89] Stephanie Weirich. RepLib: a library for derivable type classes. In *Proc. the ACM SIGPLAN Haskell Workshop*, pages 1–12, September 2006.

[90] Edwin Westbrook. Free variable types. In *Proc. the Seventh Symposium on Trends in Functional Programming (TFP'06)*, April 2006.

[91] Edwin Westbrook, Aaron Stump, and Ian Wehrman. A language-based approach to functionally correct imperative programming. In *Proc. the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, pages 268–279, September 2005.

[92] Dinghao Wu, Andrew W. Appel, and Aaron Stump. Foundational proof checkers with small witnesses. In *Proc. the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '03)*, pages 264–274, August 2003.