# High-Confidence, Modular Compiler Development in a Formal Environment

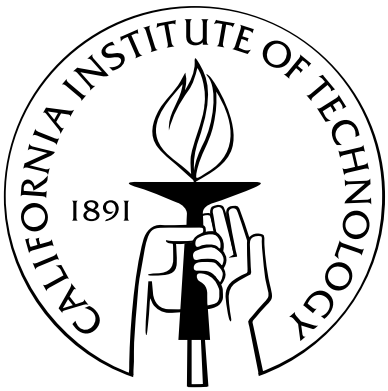
Thesis by

Nathaniel Gray

In Partial Fulfillment of the Requirements

for the Degree of

Master of Science




California Institute of Technology

Pasadena, California

2005

(Submitted June 3, 2005)

For Natale Han and Amaya Penelope.

Thanks for the big hugs, big smiles, and big diapers.

# Acknowledgements

I would like to thank my advisor, Jason Hickey, and Aleksey Nogin for their hard work and guidance. I would also like to thank Cristian Ţăpuş for his helpful comments and proofreading.

# Abstract

We present a methodology for realistic compiler development in an existing formal methods framework. Program transformations and analyses are implemented as term rewrites and inference rules, and automated proof search techniques are used to drive the compilation process. This approach allows the programmer to implement the compiler succinctly, declaratively, and modularly. We explain how our methodology separates trusted code, which can potentially corrupt compilation, from untrusted code, which cannot. We present a case study in which we have used these techniques to implement a compiler for a small ML-like programming language that produces x86 assembly code as output. We give a detailed overview of several stages of the compiler, including type inference, type checking, type erasure, CPS conversion, and closure conversion. We also describe the process of extending the minimal core compiler to include features such as integers, Booleans, operators, tuples, and recursive functions.

# Contents

# List of Figures

# Chapter 1

# Introduction

Writing reliable software is proving to be one of the most vexing challenges of the computer age. In particular, if we ever hope to build reliable software systems it is imperative that we have a reliable infrastructure upon which to build. Compilers are at the heart of the modern software infrastructure. Nearly every program in use today is itself compiled or relies on a program that is compiled. Since they play such a vital role, we would like to have a high degree of confidence that our compilers are reliable, always preserving the semantics of our programs as they are translated from high-level languages to machine code. However, compilers are large software projects, typically containing hundreds of thousands of lines of code. As in any project of such scale, errors are all but inevitable. Unfortunately, if a compiler error causes compilations to be corrupted then the effects can be widespread and the source of the problem can be hard to deduce. It therefore seems worthwhile to invest effort in developing techniques for building reliable compilers.

Ideally, our compilers would be formally verified. This would provide the highest level of confidence. Formal verification is no simple task, however. At the very least, verification requires a formal semantics for both source and target languages, and although some source languages are amenable to formal treatment very few machine architectures are. In many cases the cost of performing such a verification is prohibitively high. The benefits of verification are great, however, so even if we choose not to verify a compiler initially it would be advantageous to implement it in such a way that future verification is feasible.

In the absence of formal verification we want to employ high-confidence techniques that have been proven effective in reducing programmer errors. For example, we may want to implement our compiler in a *domain-specific language* (DSL) rather than a general-purpose language. DSLs help reduce errors in two ways: DSL solutions are generally more concise than general-purpose language solutions, so there are fewer opportunities for error; and the semantics of a DSL are usually designed such that concepts from the problem domain map directly to language constructs, reducing the chances that a domain concept will be incorrectly translated into the programming language. The latter advantage is further enhanced when a DSL is *declarative*—it allows the programmer to specify

relationships rather than procedures. Two extremely successful examples of declarative DSLs in the compiler domain are Lex [20] and Yacc [15], which have greatly reduced errors resulting from hand coding lexical analysis and parsing routines.

Another useful concept for tackling software complexity is *modularity*. Modularity brings the benefit of reducing the cognitive burden on the programmer, who only needs to understand a module's interface in order to use it. This can be critical when a project is being developed by more than one programmer, as is often the case for compilers. In modular code it is also less likely that an error in one module will cause problems in another unrelated module. Modularity is also beneficial when introducing new programmers to the project, as the amount of code they need to learn is reduced and the errors they make while learning are better isolated.

A related concept, *trust*, is a way of isolating the code that is critical to the success of the compiler. We define *trusted code* as code that has the potential to cause the compiler to produce incorrect output without generating a compile-time error, thus corrupting the compilation. This definition may seem counterintuitive due to the positive connotations of the word "trust," but in our usage it should be understood that trusted code is not code that is *trustworthy*, it is code that *must* be trusted if the compiler itself is to be trusted. In most compilers implemented using general-purpose programming languages it is difficult to separate the trusted code from the untrusted code. This is unfortunate, because if the trusted code in a compiler is verified then the compiler itself is verified. If a compiler is implemented with a clear separation between trusted and untrusted code then verification becomes much simpler. Making this distinction also isolates the code that must be examined if a compilation-corrupting error is found.

With these principles in mind, we have developed a methodology for realistic compiler development in a formal methods framework. We have also designed a compiler architecture that exploits the framework in order to provide a compiler development environment with a high degree of modularity and reliability. We have used these techniques to implement a compiler for a small ML-like programming language. Although we have not verified the compiler, we have developed it in a manner such that it is amenable to future verification.

There are many benefits to developing compilers using our techniques:

- There is a net reduction in the size of the compiler compared to traditional techniques, which reduces the opportunities for error.
- There is a clear separation between trusted and untrusted code.
- There is a *dramatic* reduction in the quantity of trusted code required in the compiler, allowing for the possibility of verifying the compiler.
- The trusted portion of the compiler is written in a concise, declarative manner that reflects a

textbook account of programming language semantics. The theorem prover effectively acts as a declarative DSL for compiler construction.

- The compiler is modular—new features can easily be added to a source language without breaking existing features.
- Due to the modularity of the compiler and the separation of trusted and untrusted code, the compiler can be verified incrementally.
- The fact that the implementation environment is a formal toolkit places the compiler in a good position for future verification efforts.

## 1.1 Traditional Methods

In this thesis we often mention "traditional methods" and "traditional compilers." We use these phrases to represent the techniques that are traditionally employed in production compilers such as the Gnu Compiler Collection [5] or the INRIA OCaml compiler [19]. In particular, the following properties are common to such implementations.

- They are implemented in procedural languages such as C, Java, or OCaml
- They are organized in terms of compiler stages
- Each compiler stage contains knowledge about every supported language feature
- They do not distinguish between trusted and untrusted code

We do not mean to imply that all existing compilers share these qualities. For example, object-oriented compiler designs sometimes use classes to achieve compositionality of features. However, we believe that most existing compilers possess at least one of these properties, and this is the way that compiler construction is taught in most university curricula today.

## 1.2 Outline

The layout of this thesis is as follows:

1. A description of MetaPRL, the logical tool that we use to build the compiler, along with a discussion of how its features are useful for compiler implementation
2. An account of the case study, including descriptions of the compiler architecture and algorithms
3. A presentation of the results of the case study

4. A discussion of future directions for this research

5. A discussion of related work

# Chapter 2

# The MetaPRL Term Language

We have developed our compiler using MetaPRL [10], an LCF-style tactic-based logical tool that uses OCaml as its tactic implementation language. It defines a syntax, based on higher-order abstract syntax (HOAS) [27], for the creation and manipulation of term trees with binding structure, which are well suited for acting as the intermediate representation (IR) of a compiler. In this chapter we discuss the following MetaPRL concepts and how they are useful for compiler implementation:

- MetaPRL *Terms*, which include bindings, for defining the IR term language
- *Sequents*, (sometimes called telescopes) for defining IR terms with unbounded arity
- *Rewrite rules* (or just *rewrites*) for performing code transformations
- *Inference rules* for defining code judgments
- Proof automation via *tactics*, which are strategies for deciding when and where to apply rewrite and inference rules

Although we have used MetaPRL in this case study it is important to point out that any logical framework with similar functionality could have been used. The techniques that we have developed can be adapted to any sufficiently powerful logical platform.

## 2.1 Terms and Sequents

Every compiler needs an intermediate representation (IR) for programs as they are compiled. In our compiler, MetaPRL terms are used for this purpose. A simple term has three components: 1) an operator-name (like "sum"), which is a unique name identifying the kind of term; 2) an optional list of integer or string parameters; and 3) an optional list of subterms, each with zero or more variable bindings. We use the following notation to describe terms:

$$\underbrace{opname}_{operator\ \ name}\ \underbrace{[p_1;\cdots;p_n]}_{parameters}\underbrace{\{\overrightarrow{v}_1.t_1;\cdots;\overrightarrow{v}_m.t_m\}}_{subterms}$$

The $\overrightarrow{v}_i$ are comma-delimited vectors of variable bindings. All the free occurrences of $\overrightarrow{v}_i$ in $t_i$ are bound by the operator. When $n = 0$ or $m = 0$, the corresponding brackets or braces may be omitted; when $\overrightarrow{v}_i$ is empty, the dot before $t_i$ is usually omitted. The parameters $p_i$ are constants.

Each operator has a fixed arity, which includes a fixed number of parameters, a fixed number of subterms and a fixed number of bindings for each subterm. If two operators have different arities, they are considered to be distinct even if they have the same opname.

The term language is untyped and terms carry only structural information. It is up to the programmer to define the semantics of terms. Below are a few examples of terms that could be used in a formalization of a simple typed lambda calculus, along with pretty-printed forms. (We generally use pretty-printed forms for terms.)

| Term | Pretty-printed form |
|---|---|
| `TyInt` | $\mathbb{I}$ |
| `integer[1]` | 1 |
| `apply{'f; 'a}` | $f(a)$ |
| `lambda{'t; v. apply{'f; 'v}}` | $\lambda\, v{:}t.\ f(v)$ |
| `TyFun{'x; 'y}` | $x \to y$ |

The simplest terms, like `TyInt`, are just opnames. Numbers have a constant integer parameter. The `lambda` term contains a binding occurrence: the variable `x` is bound in the subterm `apply{'f; 'x}`. Variable references are preceded by an apostrophe to distinguish them from bare opnames. There are two kinds of variables: first-order variables and second-order variables (sometimes abbreviated FO and SO variables, respectively). First-order variables are created by explicit bindings, like `x` in the example above. They represent simple, object-language-level variables.

Variables that appear unbound in a term are second-order variables [25]. Both of the variables in `TyFun{'x; 'y}` are second-order. A term containing one or more second-order variables is a *term scheme*—a pattern that can be matched to specific term instances. So, for example, `TyFun{TyInt; TyInt}` is a term that matches the `TyFun{'x; 'y}` scheme. The pattern `!v` can be used instead of `'v` to match a first-order variable instead of an arbitrary subterm.

The scope of first-order variables within term schema can be controlled by placing square brackets after the second-order variables. If a first-order variable appears in the square brackets after an SO variable it can appear free in the subterm matched by that SO variable. If it is left out of the square brackets it is not allowed to appear free when matching that subterm. Omitting the square brackets is equivalent to including empty brackets. For example, a generic term scheme for untyped function definitions would be `lambda{v. 'e['v]}`. If we mistakenly used `lambda{v. 'e}` instead then the scheme would only match functions that never reference their arguments. We discuss matching schema further in Section 2.2.

The term language as described so far is complete and quite useful, but the fixed arity requirement

for terms can be a bit inconvenient. We may, for example, wish to define functions as having multiple arguments that cannot be partially applied. We could use a curried representation, but it would require some work to make sure that partial application was not possible. A better solution is to use a type of term that supports variable arity. MetaPRL provides *sequents* (sometime called telescope terms), which can contain a variable number of bindings. Although sequents are usually interpreted as defining judgments, and indeed this is how they are used in the meta-language, this is not their only possible interpretation, and we are free to exploit them in our IR.

The concrete and pretty-printed syntax for sequents is given below.

| Concrete syntax | Pretty-printed form |
|---|---|
| `sequent [a]{ v1:t1; ... ; vn:tn >- c }` | $a\{\!\lbrack v_1 : t_1; \ldots; v_n : t_n \vdash c \rbrack\!\}$ |

The term $c$ is the *conclusion* of the sequent and the terms $t_i$ are its *hypotheses*. Note that the length of the hypothesis list can change, and it can be empty ($n = 0$). The variables $v_i$ introduce binding occurrences; each $v_i$ is bound in all $t_j$ for $j > i$, and in $c$. Finally, the term $a$ is the *sequent argument*, which specifies what kind of sequent it is—the sequent argument plays essentially the same role for sequents as the operator name plays for ordinary terms. The `v:` variable binding may be omitted from a hypothesis if it is not used, and the `>- c` can also be omitted if the sequent represents a simple list of terms. The $t_i$ terms are often, but not always, interpreted as the types associated with the $v_i$ variables. Tuples are represented as $\texttt{tuple}\{\!\lbrack t_1; \ldots; t_n \vdash \rbrack\!\}$, for example.

*Sequent schema* [25] may also include *context* meta-level variables that stand for arbitrary lists of hypotheses. For example, the sequent scheme

$$a\{\!\lbrack \Gamma;\, v : T;\, \Delta[v] \vdash c[v] \rbrack\!\}$$

(where $\Gamma$ and $\Delta$ are context variables and $T$, $a$, and $c$ are second-order variables) matches any sequent with at least one hypothesis. Note the square brackets after $\Delta$; scoping of FO variables within context variables is specified just as in SO variables. Context variables themselves are scoped as well, but they are treated with the opposite assumption from first-order variables. That is, the variables that are bound in the hypotheses that a context variable matches are allowed to appear free in any subterm unless the programmer explicitly specifies otherwise. We do not describe the syntax for writing such specifications because we do not need them in this thesis.

Using sequents, we can easily represent multi-argument functions. The untyped function of two arguments $\lambda(f, x).f(x)$ can be implemented as

$$\lambda\{\!\lbrack f : T_u;\, x : T_u \vdash f(x) \rbrack\!\}$$

where $T_u$ is a type place holder for untyped terms. When defining the function type $(T_1, T_2, T_3) \to T$

we can omit the variable bindings (assuming our type system is not dependent):

$$\texttt{TyFun}\{\!|\, T_1; T_2; T_3 \vdash T \,|\!\}$$

When using sequents to represent terms with lists of bindings it is convenient to use vectors to denote contexts. We also use more specialized pretty-printing for terms and types when they have well-established notational conventions. Here are a few examples of our vector notation:

| Sequent Notation | Vector Notation |
|---|---|
| $\lambda\{\!\mid \Gamma \vdash e \mid\!\}$ | $\lambda\, \overrightarrow{v{:}T}.\ e$ |
| $\texttt{TyFun}\{\!\mid \Gamma \vdash T \mid\!\}$ | $\overrightarrow{T_1} \to T$ |

Note that vector notation carries slightly more information than the generic context notation. We specify whether the bound variables, the associated terms, or both are significant. However, it is important to remember that the "variables" that appear in vectors are not first-order variable bindings! The vector form $\overrightarrow{v{:}T}$ in its entirety represents a single context variable. As mentioned earlier, context variables are allowed to appear free in any subterm that doesn't specify otherwise.

We sometimes mix vector and scalar notation to specify sequent schema that match terms with a specific structure. For example, $\lambda\,(v_1{:}T_1, \overrightarrow{v{:}T}).\ e[v_1]$ matches any function with at least one argument, binding the first argument to $v_1$, its type to $T_1$, and the rest of the argument list to $\overrightarrow{v{:}T}$. Notice that $\overrightarrow{v}$ does not need to be listed explicitly in the square brackets after $e$.

## 2.2 Term Rewrites

A term rewrite specifies the *bidirectional* equivalence of two term schemas. Any term that matches the left-hand-side of the rewrite (its *redex*) can be replaced with the corresponding value of the right-hand-side of the rewrite (its *contractum*), and vice-versa, in any context. For example, $\beta$-reduction could be specified with the following rewrite.

$$(\lambda x.e_1[x])\ e_2 \leftarrow [\textsc{beta}] \rightarrow e_1[e_2]$$

By replacing $e_1[x]$ on the left-hand side with $e_1[e_2]$ on the right hand side we have specified that the rewrite should substitute $e_2$ for all occurrences of $x$ in $e_1$. The rewriter $\alpha$-renames $e_2$ as necessary to ensure that no variables are captured during this substitution. It is also possible to perform simultaneous substitution of multiple variables:

$$(\lambda(x, y).e_1[x, y])\ (e_2, e_3) \leftarrow [\textsc{beta2}] \rightarrow e_1[e_2, e_3]$$

Rewrites can either be *primitive* or *derived* from other rewrites. Primitive rewrites are accepted

by the rewriter as axiomatic—it is entirely up to the programmer to ensure their correctness. In order to reduce opportunities for error, we would like to minimize the number of primitive rewrites. For transformations that need not be axiomatic, derived rewrites are available. The programmer can use the interactive MetaPRL environment to prove that derived rewrites can be expressed in terms of other rewrites and then save these proofs so that they can be re-checked when revisions are made to the code. Derived rewrites provide us with the ability to extend the compiler with optimized transformations without increasing the size of the compiler's trusted code base.

One advantage of using rewrites for code manipulation is that MetaPRL's rewrite engine provides protection against a variety of syntactic errors. For example, it is impossible to specify a rewrite that introduces a free variable or causes a variable to be captured. While this is normally desirable, it can be challenging to formulate certain code transformations without using temporary free variables or exploiting variable capture. However, we have not had trouble working out suitable formulations for the transformations we have investigated.

## 2.3 Judgments and Inference Rules

MetaPRL is a theorem prover, and as such it has strong support for specifying logical judgments and inference rules. In our compiler we use these facilities to implement type checking and to drive the overall compilation process.

The compilation process is expressed in MetaPRL as a judgment of the form $\Gamma \vdash \langle\langle e \rangle\rangle$, which states that the program $e$ is compilable in the logical context $\Gamma$. The exact meaning of the $\langle\langle e \rangle\rangle$ judgment is defined by the target architecture. A program $e$ is compilable if it can be represented by a sequence of valid assembly instructions $e'$. The compilation task is a process of proving that, given the rules and rewrites we specify as primitive, the source program $e$ can be rewritten to an equivalent assembly program $e'$. Each stage of the compiler forms a phase of this proof. Different stages define different judgments that must be satisfied for the proof to succeed.

For example, type checking defines a well-typed judgment $\Gamma \vdash e \in T$. To allow proofs of well-typedness we define a logic of type checking rules, specified as meta-implications. The concrete syntax for the lambda typing rule of a simply typed lambda calculus is:

```
sequent{ <H> >- 'ty_x = 'ty_arg } -->
sequent{ <H>; 'x in 'ty_arg >- 'e['x] in 'ty_ret } -->
sequent{ <H> >- lambda{ 'ty_x; x. 'e['x] } in TyFun{'ty_arg; 'ty_ret} }
```

where `<V>` is the concrete syntax for a context variable $V$. Here, MetaPRL sequents are employed to represent logical sequents. In this setting, contexts represent *type environments*, which contain bindings of variables to types (or, if you will, variable well-typedness propositions) and assertions of type well-formedness. The concrete syntax above is equivalent to the following rule in traditional

sequent calculus notation:

$$\frac{\Gamma \vdash t_x = t_a \quad \Gamma; x \in t_a \vdash e[x] \in t_r}{\Gamma \vdash \lambda\, x\!:\!t_x.\ e[x] \in t_a \to t_r}$$

The judgment below the line is called the *goal* of the rule and those above the line are referred to as its *premises*. Proofs are normally performed by *backward-chaining* of inferences—the proof tree is built upwards from the goal. When applying an inference rule during a backward-chaining proof, the premises introduce new *subgoals* that must be proven in order to prove the goal.

Like rewrites, inference rules can be primitive or derived from other rules. To keep the set of rules sound it is desirable to use a minimal set of primitive rules and derive others as needed for efficiency or expressiveness.

One important fact to understand about our compiler is that rewrites and inference rules are the *only* mechanism by which code is modified. This means that if all of the rewrites and rules in the compiler preserve semantics then the compiler itself must be correct! This clear separation between trusted and untrusted code is a very powerful benefit derived from the formal framework. Furthermore, the trusted code is written as fragments of declarative domain-specific code. Rules and transformations are often short and simple enough, and close enough to their textbook equivalents, that simple inspection is sufficient for one to believe that they are correct.

## 2.4 Tactics

Rewrites and inference rules are declarative tools for specifying transformations and logical implications, but they do not specify when or where they should be used. If we possessed a nondeterministic machine on which we could run MetaPRL then this would not be a problem. We could simply specify all the rules and rewrites of our system and nondeterministically try every possible compilation. Assuming our rules and rewrites preserve the semantics of the language, we could choose any complete compilation that suited us. Lacking such a machine, however, we need a way to specify which rules and rewrites to apply, where to apply them, and in what order.

These decisions are handled by LCF-style *tactics* [6], or *guidance code*. MetaPRL uses OCaml as its tactic language, providing a number of built-in proof automation tactics as well as the primitives required for the user to build his own. For example, the following tactic might be employed to search for redices in a program and $\beta$-reduce them:

```
let beta_reduce_all_subterms = rw (sweepDnC (repeatC beta)) 0
```

Assume that `beta` identifies the $\beta$-reduction rewrite from Section 2.2, and also note that rewrites have type `conv`. A function that takes a `conv` and returns a `conv` is called a *conversional*. The other elements of this tactic are provided by the system:

| | |
|---|---|
| `rw: conv -> int -> tactic` | A tactic that applies the given conversion to the subgoal identified by the integer (subgoal 0 is the goal). |
| `sweepDnC: conv -> conv` | A conversional that applies its argument to the current term and then recursively applies itself to every subterm. |
| `repeatC: conv -> conv` | A conversional that applies its argument repeatedly until it either fails or reaches a fixed point. |

In summary, this tactic starts at the outermost term of the current proof goal and repeatedly applies the `beta` rewrite until it fails or makes no progress. Then it does the same on each subterm recursively.

One very nice feature of MetaPRL relating to the implementation of tactics is that it allows us to embed MetaPRL term syntax in OCaml tactic code. Terms can be constructed and destructed using *quotations* and *antiquotations* respectively. Quotations build MetaPRL terms, potentially using OCaml values, while antiquotations allow us to match terms and introduce their variables and subterms into the OCaml environment. This example introduces their concrete syntax:

```
let x = 1 in
let term = <:con< Apply{'f; Integer[$x$]} >> in (* a quotation *)
match term with
   << Apply{'f; 'x} >> -> (* An antiquotation *)
       let y = 10 + (int_of_mp_integer x) in
       . . .
 | << Lambda{ x. 'e } >> -> . . .
```

In our examples, we use pretty-printed syntax, underlining variable references in quotations. Also, when we reference a variable defined in an antiquotation we use a mathematical typographic style identical to the style within the antiquotation:

```
let x = 1 in
let term = ▷ f(x) ◁ in (* a quotation *)
match term with
   ◁ f(x) ▷ -> (* An antiquotation *)
       let y = 10 + (int_of_mp_integer x) in
       . . .
 | ◁ λx.e ▷ -> . . .
```

Note that it is not necessary to specify the scope of FO variables in quotations or antiquotations.

## 2.5 Trusted and Untrusted Code

Tactics are powerful, but their power is limited to the domain of proof guidance. In particular, they can only affect the state of the compilation by invoking rewrites or proof rules. Tactics can only transform the program within the space defined by the nondeterministic machine described above. This limitation has consequences that are highly beneficial from the standpoint of reliability,

for if our rewrites are semantics-preserving and our rules are semantically valid *it is impossible for tactic code to corrupt a compilation*[1]. This means that if we are careful when writing our rules and rewrites, our tactic code does not have to be trusted for the compiler to be trusted. There is a clear separation between trusted and untrusted code: tactics are untrusted, while inference rules and rewrites are trusted.

The desire to ensure that we write semantically valid rules and rewrites (and to keep them *obviously* valid, wherever possible) is one of the primary design principles of our methodology, but in this early version of the compiler we do not adhere to it in every case. We have sometimes compromised on semantical validity when it would require significant effort to solve the semantic problem satisfactorily. For example, because rewrites are bidirectional and context-free, strict semantics-preservation requires us to preserve all information—all free variables that appear in the redex of a rewrite must also appear in its contractum. This means that the naïve typed $\beta$-reduction rewrite is not semantics preserving.

$$(\lambda x{:}T.e_1[x])\ e_2 \leftarrow [\textsc{Beta-T}] \rightarrow e_1[e_2]$$

The problem with Beta-T is that it could be applied in reverse with an arbitrary type provided for $T$, creating an ill-typed term from a well-typed one. One way to solve this is to add a type constraint to $e_2$ in the contractum. This works, but the program would quickly become polluted with type constraints. What we really want is a way to define *unidirectional* rewrites. Later versions of MetaPRL have added this functionality, so this problem has been solved. However, it is exceedingly unlikely that a programmer would *accidentally* use Beta-T in reverse, so it would not be a great cause for concern to include such a rewrite until a better solution became available.

Although the inclusion of invalid rewrites weakens the separation between trusted and untrusted code, it does not destroy it. The occurrences of such rewrites are few, well-understood, and limited in impact. The use of these rewrites means that in this version of the compiler there is a small body of tactic code that we must trust to uphold certain invariants. The invariants in question are very simple, however. For example, we must trust that tactics do not apply rewrites in reverse, we must trust that certain stages are applied in order, and we must trust that rewrites from one stage are not applied during another stage. These invariants are unlikely to be accidentally violated. In more recent versions of the compiler most of these invalid rewrites have been eliminated, so the separation between trusted and untrusted code is almost completely clean.

[1] In this context, a corrupt compilation is one that appears to have succeeded but has actually produced output that is not semantically equivalent to its input.

## 2.6 Summary

Before we describe the application of these techniques to a real compiler, let us summarize the concepts we have described in this chapter.

- We represent programs with MetaPRL terms, which contain first-order variable bindings.
- Term schema are patterns that match terms. Second-order variables are used to match arbitrary subterms. When a first-order variable binding appears in a term scheme, its scope must be explicitly specified for any subterms.
- Sequents are terms that contain a list of hypotheses (variable bindings and associated terms) and a single goal. The list can have arbitrary arity.
- Context variables, which we represent as vectors, are patterns that match zero or more hypotheses in a sequent.
- Term rewrites are used to transform one term to another. Rewrites represent bidirectional equivalences, and are specified using pairs of term schema.
- Logical judgments and inference rules are used to define the compilation process and also to perform analyses such as type checking.
- Rewrites and inference rules are the *only* mechanisms for program transformation in our compiler.
- Tactics are used to decide which rules and rewrites to apply, where to apply them, and in what order.
- If the system's rules and rewrites are semantically valid then it is impossible for tactic code to corrupt a compilation, though errors in tactics can cause compilation to fail. Most, but not all, of the rules and rewrites in the compiler we describe are semantically valid.

# Chapter 3

# Case Study

In this chapter we describe the implementation of a compiler for a small ML-like language using the primitives described in Chapter 2. We begin by describing the compiler's architecture, then present a more detailed description of the compiler stages. Finally, we discuss the language features we have added as extensions to the core compiler.

## 3.1 Architectural Overview

Like most compilers, ours is divided into stages. Typical stages include type inference, continuation passing style (CPS) transformation, closure conversion, and various optimizations. Unlike most compilers, however, our compiler is structured as a minimal core compiler that can be extended to support various language features. The core compiler supports compilation of a small polymorphic lambda calculus that does not even include integers or Booleans. All additional functionality is provided in extensions. A diagram of the architecture can be found in Figure 3.1. Note the shaded boxes that illustrate file boundaries.

Each core compiler stage is implemented in one file. Extensions, on the other hand, generally

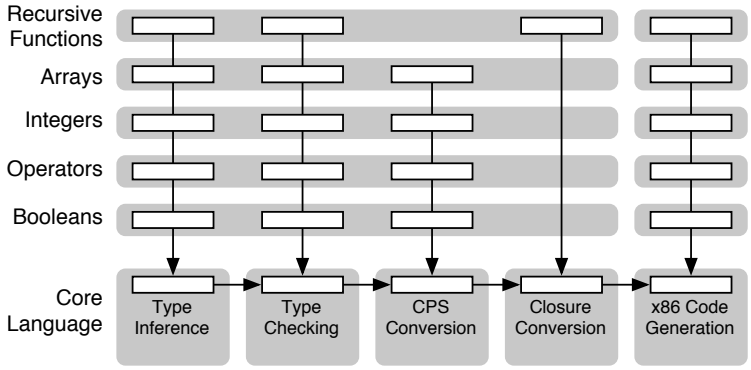



Figure 3.1: The Compiler Architecture. Shaded regions correspond to file boundaries.

combine *all* of their code into one or two files. This compositional structure, which isolates features from each other and the core compiler, has several nice properties for language experimentation. To understand the basic functionality of a compositional stage one needs only to understand how it operates on the core language. This learning process is simplified because there are no additional features to distract the programmer. The architecture also simplifies the process of adding new stages, as they can be developed and tested for the core language first and later extended with any necessary feature support.

In addition, this layout makes it easy to understand how a given feature works, since the code for that feature is isolated and contained in a small number of files. There are typically two files per feature: one for the front end support and one for the back end support. This is possible because each feature adds very little code to any one stage—often just one or two rewrites. Some features do not need to add *any* code to some stages. For example, the recursive function extension is the only feature that needs to extend the closure conversion stage. The number of files per extension will, of course, increase as we begin to support more back ends, but all of the files will remain together in their own directory, isolated from other features.

In contrast, consider a compiler organized by compiler stages. The code for each stage contains references to every language construct. To understand any stage you must understand how it works for the entire language, not just a minimal core language. In addition, if you want to add a new feature to the language you usually end up editing almost every file in the compiler. To be fair, this layout is potentially advantageous if the language being compiled does not undergo many changes but new optimizations are frequently added. Our layout simplifies the process of adding language features at the expense of making it slightly more complicated to add compiler stages. We feel that this is a good design decision for a research compiler intended for language experimentation.

While there are obvious advantages to our type of layout, one may suspect that it may mask troublesome interactions between features. This is a danger, but the logical toolkit makes it simple to guarantee orthogonality of features. If an extension only refers to opnames that it defines then it will be guaranteed to be compositional.

## 3.2 Stages of the Compiler

We now discuss the stages of the compiler, along with the languages and abstractions used to represent the program being compiled during the various stages. The major stages of the compiler and their corresponding languages are described in Figure 3.2.

| Compiler Stage | Input language | Output language |
|---|---|---|
| Parsing | Source language | AST |
| Type Inference | AST | Typed AST (TAST) |
| Type Checking | TAST | TAST |
| CPS Conversion | TAST | TAST |
| Closure Conversion | TAST | TAST |
| Code Generation | TAST | x86 assembly |

Figure 3.2: Compiler Stages and Languages

| | | |
|---|---|---|
| $e ::=$ | $v$ | Variables |
| $\mid$ | **let** $f(v_1, \cdots, v_n) = e_1$ **in** $e_2$ | Function Definition |
| $\mid$ | $f(e_1, \cdots, e_n)$ | Function Application |
| $\mid$ | **let** $v = e_1$ **in** $e_2$ | Let-Abstraction |
| $\mid$ | $e : T$ | Type Constraint |

Figure 3.3: The Core Source Language

### 3.2.1 The Core Source Language

The specifics of the source language are not central to our work. We want to make sure that the language is *realistic* in its demands from the compiler, but we are more interested in investigating our compiler architecture than we are in creating a new language. Thus, we have chosen a source language that is small but reasonably complete, derived from ML.

The source language is very similar to System F, but functions may take multiple arguments and cannot be partially applied. The language is divided into a *core* language and *extension* features. In this section we only discuss the core language. Extensions are described as they are introduced in Section 3.3. The syntax of the core source language is described in Figure 3.3.

At first glance it may appear that we have oversimplified our language. It has no loops, no recursion, and because type inference must be possible it is not possible to define the Y combinator. Thus, our core language is not even Turing complete! However, this is only the core of the language. All of these capabilities are added compositionally by extensions.

### 3.2.2 Parsing

The parser is responsible for converting the source language into an *abstract syntax tree* (AST). The parser is implemented using the Phobos extensible parser tool [7], developed as part of the Mojave project [14]. We do not go into detail about Phobos, referring the reader to the published literature.

The syntax for the untyped AST is described in figure 3.4. It is almost a one-to-one translation of the source language, except that functions are represented as values without scope.

| | | | |
|---|---|---|---|
| $e$ | ::= | $v$ | Variable |
| | \| | $\lambda_u \overrightarrow{v}.\ e$ | Function Definition |
| | \| | $f(\overrightarrow{e})_u$ | Function Application |
| | \| | $\mathbf{let}_u\ v = e_1\ \mathbf{in}\ e_2[v]$ | Let-Abstraction |
| | \| | $e : T$ | Type Constraint |

Figure 3.4: The Untyped Abstract Syntax Tree Language

| | | | |
|---|---|---|---|
| $e$ | ::= | $v$ | Variable |
| | \| | $\lambda \overrightarrow{v{:}T}.\ e$ | Function Definition |
| | \| | $f(\overrightarrow{e} : \overrightarrow{T})$ | Function Application |
| | \| | $\mathbf{let}\ v{:}T = e_1\ \mathbf{in}\ e_2[v]$ | Let-Abstraction |
| | \| | $\Lambda\ \overrightarrow{\alpha}.\ e$ | Type Abstraction |
| | \| | $\texttt{TyApply}\{e{:}T_e;\ \overrightarrow{T}\}$ | Type Application |
| | \| | $\texttt{Constrain}\{e{:}T\}$ | Type Constraint |

Figure 3.5: The Typed Abstract Syntax Tree Language

### 3.2.3 Type Inference

After parsing, the compiler must perform type inference on the untyped AST. The result is a typed AST (TAST). The TAST syntax is given in Figure 3.5. The type system is a standard ML-style system [9], defined by the grammar in Figure 3.6. Although it is the first stage we examine in detail, type inference is exceptional in our compiler because its implementation is almost entirely informal. The tactic portion of type inference is a compositional version of algorithm W [3] implemented in OCaml. The algorithm takes five arguments as input:

| | |
|---|---|
| `infer` | A recursive reference to the inference function |
| `tenv` | The set of type variables defined in this expression |
| `venv` | The variable → type environment for this expression |
| `s` | The substitution environment for this expression |
| `e` | The expression whose type is to be inferred |

It returns a triple: `(e', s', T_e')`, where `e'` is the typed version of `e`, `s'` is the new substitution environment, and `T_e'` is the type of `e'`.

For variables, the type is simply looked up in `venv`.

| | | | |
|---|---|---|---|
| $T$ | ::= | $\bot$ | Void type |
| | \| | $\top$ | Top type |
| | \| | $\alpha$ | Type variables |
| | \| | $\overrightarrow{T} \to T$ | Multi-argument function types |
| | \| | $\forall\, \overrightarrow{\alpha}.\ T$ | Universal abstraction |

Figure 3.6: The Core Type System

```
let infer_lambda infer tenv venv s ◁λ_u v⃗. e▷ =
    let venv, tenv, args, argtypes =
        Fold for each v in v⃗:
            let a = A fresh type variable in
            let venv, tenv = Update environments with v:a in
                venv, tenv, (v:a)::args, a::argtypes
    in
    let e', s', ty_e' = infer tenv venv s e in
        ▷λ args. e'◁, s', ▷argtypes → ty_e'◁

let infer_apply infer tenv venv s ◁f(e⃗)_u▷ =
    let a = A fresh type variable in
    let f', s', t_f = infer tenv venv s f in
    let e_t, s', types =
        Fold for each e in e⃗:
            let e', s', t = infer tenv venv s' e in
                e'::e_t, s', t::types
    in
    let s' = unify s' t_f (▷types → a◁) in
        ▷f'(e_t : types)◁, s', ▷a◁
```

Figure 3.7: Type inference for functions and applications. We use the informal notation *Fold for each $x$ in $\overrightarrow{x}$* to represent the `List.fold_right` function. `List.fold_right f [a1; ...; an] b` is equivalent to `f a1 (f a2 (... (f an b) ...))`.

```
let infer_var infer tenv venv s ◁!v▷ =
    try let ty = SymbolTable.find venv v in
        ▷v◁, s, ▷ty◁
    with Not_found -> Raise an unbound variable exception
```

Type inference of `Constrain` terms is also simple. We infer the type of the constrained expression and add the constraint to the substitution environment.

```
let infer_constrain infer tenv venv s ◁Constrain{e:T}▷ =
    let e', s', t2 = infer tenv venv s e in
    let s' = Add constraint T=t2 to s' in
        ▷Constrain{e':T}◁, s', ▷T◁
```

To infer the type of a function, we create fresh type variables for each of its parameters, then infer the type of its body. Inference of applications uses the `unify` function provided by MetaPRL, which takes a substitution and two terms and extends the substitution to unify the terms (if possible). The type of each function argument is unified with the corresponding parameter type in the function's type, and a fresh type variable is created to unify with the return type. The code for inference of functions and applications appears in figure 3.7.

When inferring the type of a let expression, we generalize any free type variables. We anticipate that the core language will be extended with imperative features, so we enact a *value restriction*—we only generalize *syntactic values*. For now, our definition of syntactic value includes only lambda expressions. This is overly restrictive, but known to be safe. In later versions of the compiler we have

relaxed this definition. MetaPRL provides the `free_vars_set` function for calculating the set of free variables in a term. Because HOAS requires many free-variable calculations, the implementation of MetaPRL terms is designed to make this calculation efficient.

```
let infer_let infer tenv venv s ◁(letᵤ v = e₁ in e₂)▷ =
    let e1, s', ty_e1 = infer tenv venv s e₁ in
    let e1, ty_e1 =
        if not is_value a then
            e1, ty_e1
        else
            let w = free_vars_set ty_e1 in
            let w = Remove any free vars defined in s' in
            if is_empty w then
                e1, ty_e1
            else
                (▷ Λ w. e1 ◁), (▷ ∀ w. ty_e1 ◁)
    in
    let venv = SymbolTable.add venv v ty_e1 in
    let e2, s', ty_e2 = infer tenv venv s' e₂ in
        (▷ let v:ty_e1 = e1 in e2 ◁), s', ▷ ty_e2 ◁
```

Since type inference is a *syntax-directed* transformation, we can get special support from MetaPRL for combining these individual functions into a unified type inference tactic. A MetaPRL *resource* is a structure that has been designed to simplify this task. A resource is a tactic that dispatches sub-tactics based on extensible pattern matching against terms. After declaring and initializing the type inference resource `typeinf` we can build the tactic by specifying pairs of antiquotations and tactics.

```
let resource typeinf +=
   [◁e▷, report_error;
    ◁!v▷, infer_var;
    ◁Constrain{e:T}▷, infer_constrain;
    ◁λᵤ v⃗. e▷, infer_lambda;
    ◁e₁(e₂)ᵤ▷, infer_apply;
    ◁letᵤ v = e₁ in e₂▷, infer_let]
```

When the `typeinf` tactic is applied to a term, the tactic corresponding to the best (most specific) match in the table is applied. Because of the best-match semantics, the `report_error` tactic is only applied if no other pattern in the table matches. New match cases can later be added by extensions, and the antiquotations in the table can include nested terms if necessary.

The formal portion of type inference is defined by a single judgment:

$$\frac{\vdash \langle\langle \texttt{Constrain}\{e{:}T\}\rangle\rangle}{\vdash \langle\langle \mathit{erase}\{e\}\rangle\rangle}\ \text{C-Infer}$$

This judgment may appear somewhat backwards, since it mentions type erasure ($\mathit{erase}\{e\}$, described in Section 3.2.4) and type constraint but not type inference. This rule defines equivalence between typed and untyped programs. It says that a typed program is equivalent to an untyped one (in

$$
\begin{array}{ll}
\mathit{erase}\{\texttt{Constrain}\{e{:}T\}\} & \mathit{erase}\{\textbf{let } v{:}T = e_1 \textbf{ in } e_2[v]\} \\
\quad \leftarrow [\text{E-CONSTRAIN}] \rightarrow & \quad \leftarrow [\text{E-LET}] \rightarrow \\
\texttt{Constrain}\{\mathit{erase}\{e\}{:}T\} & \textbf{let}_u\ v = \mathit{erase}\{e_1\} \textbf{ in } \mathit{erase}\{e_2[v]\} \\
\\
\mathit{erase}\{f(\overrightarrow{e} : \overrightarrow{T_e})\} & \mathit{erase}\{\lambda\, \overrightarrow{v{:}T}.\ e\} \\
\quad \leftarrow [\text{E-APPLY}] \rightarrow & \quad \leftarrow [\text{E-}\lambda] \rightarrow \\
f(\mathit{erase}\{\overrightarrow{e}\})_u & \lambda_u\, \overrightarrow{v}.\ \mathit{erase}\{e\} \\
\\
\mathit{erase}\{\Lambda\, \alpha_1, \ldots, \alpha_n.\ e[\alpha_1, \ldots, \alpha_n]\} & \mathit{erase}\{\texttt{TyApply}\{e{:}T_e;\ \overrightarrow{T}\}\} \\
\quad \leftarrow [\text{E-}\Lambda] \rightarrow & \quad \leftarrow [\text{E-TYAPPLY}] \rightarrow \\
\mathit{erase}\{e[\top, \ldots, \top]\} & \mathit{erase}\{e\}
\end{array}
$$

$$\mathit{erase}\{!v\} \leftarrow [\text{E-VAR}] \rightarrow !v$$

Figure 3.8: Type Erasure Rewrites

the sense of the compilability judgment) if 1) the typed program is well-typed, and 2) erasing the types from it yields the untyped program. This rule does not specify *how* the typed program was produced—it could have been produced by an oracle, or even by hand. Rather than trusting the entity that gave us the typed program to produce valid output, we validate the program for any properties that we care about.

But how do we actually *use* the C-Infer rule? Parsing yields an untyped expression $e_u$ that we wish to compile. In other words, we want to prove the judgment $\vdash \langle\!\langle e_u \rangle\!\rangle$. To do this, we first execute the informal type inference algorithm to produce a typed expression $e_i$ and its type $T_i$. We then apply a logical *cut* rule to insert $\langle\!\langle \mathit{erase}\{e_i\} \rangle\!\rangle$ into our proof.

$$\frac{\langle\!\langle \mathit{erase}\{e_i\} \rangle\!\rangle \vdash \langle\!\langle e_u \rangle\!\rangle \quad \vdash \langle\!\langle \mathit{erase}\{e_i\} \rangle\!\rangle}{\vdash \langle\!\langle e_u \rangle\!\rangle}\ \textsc{Cut}(\langle\!\langle \mathit{erase}\{e_i\} \rangle\!\rangle)$$

Note that this is not a new axiom, but an application of an existing axiom. If erasing the types of $e_i$ yields $e_u$ then the first subgoal is proved. We then can attempt to prove the second subgoal by applying the C-Infer rule.

The use of an informal type inference algorithm runs somewhat counter to our goal of using declarative implementation as much as possible. We have chosen to validate a typed program from an untrusted source instead of producing the program with trusted code in the first place. Our motivation for taking this route was a belief that the complexity of a formal type inference implementation would likely outweigh the benefits it would bring. We do believe a formal implementation would be possible, however, so we may attempt one in the future.

### 3.2.4 Type Erasure

As we have seen, type inference relies a great deal on informal code. We need to verify that this step has not changed the program in any way other than adding type information. By referencing $erase\{e\}$ instead of $e_u$ directly, the C-INFER rule forces the compiler to verify that erasing the types from $e$ yields a term that is alpha-equivalent to $e_u$. The $erase\{\}$ term itself is a meta-language operator that must be eliminated by the end of the erasure stage. Introducing meta-language constructs into programs in order to transform them is one of the principle techniques used in our compiler. Type erasure and many other transformations are *sweep-down* transformations. In a sweep-down transformation the meta-operator is introduced at the top level of the program. For any given term the operator is applied to the term itself and then recursively applied to its subterms. When the operator hits a leaf (variable) term it is discarded.

One might wonder what would happen if a meta-operator such as $erase\{\}$ was left in a program, perhaps because of a faulty tactic. It may appear that this is a way for a tactic to corrupt a compilation. This is not the case, however. Later stages of the compiler will not have any rules or rewrites that deal with the meta-operator. The code generation stage, for example, will not have any rules for generating code from an $erase\{\}$ operator. As soon as such a stage is reached the compilation will fail. Thus, an error of this sort *can* cause the compiler to fail, but *cannot* cause the compilation to be corrupted.

The rewrites for type erasure are given in figure 3.8. Before discussing the semantics of the rules there is a notational issue that should be mentioned. We use vector notation in this thesis to present the large-step semantics of various stages. The actual implementations of vector rewrites are generally more fine-grained. For example, the E-$\lambda$ rule in figure 3.8 is actually composed of three rules:

$$\begin{array}{ll} erase\{\lambda\, \overrightarrow{v{:}T}.\ e\} & Erase\{\!|\ \overrightarrow{v_1} \vdash \lambda\,(v{:}T, \overrightarrow{v_2{:}T_2}).\ e[v]\ |\!\} \\ \quad \leftarrow [\text{E-}\lambda\text{-START}] \rightarrow & \quad \leftarrow [\text{E-}\lambda\text{-CONS}] \rightarrow \\ Erase\{\!|\ \vdash \lambda\, \overrightarrow{v{:}T}.\ e\ |\!\} & Erase\{\!|\ \overrightarrow{v_1}; v \vdash \lambda\, \overrightarrow{v_2{:}T_2}.\ e[v]\ |\!\} \end{array}$$

$$Erase\{\!|\ \overrightarrow{v} \vdash \lambda\,().\ e\ |\!\} \leftarrow [\text{E-}\lambda\text{-NIL}] \rightarrow \lambda_u\, \overrightarrow{v}.\ erase\{e\}$$

This is a common idiom for processing sequents in sweep-down transformations such as $erase\{\}$. The sequent being transformed is first wrapped in an outer sequent with a temporary sequent argument (in this case, *Erase*). Each hypothesis of the inner sequent is processed and its binding is moved to the outer sequent. When the inner sequent has no more hypotheses it is discarded, the outer sequent is replaced by the transformed version of the inner sequent (in this case $\lambda_u$), and the transformation recurses into the conclusion.

For the most part, the $erase\{\}$ stage is quite simple. Typed terms are just replaced by their untyped counterparts, or discarded if there are no such counterparts. `Constrain`$\{e : T\}$ is the

exception, since any type constraints appearing in $e$ must have been put there by the programmer. The whole transformation is syntax-directed so we use a MetaPRL resource to dispatch the rewrites.

The E-$\Lambda$ rewrite is worth examining in a bit more detail. This is a case where HOAS constrains us somewhat. We would like to simply write $\mathit{erase}\{e\}$ on the right-hand side and forget about the type variables since we know that they are all about to be erased anyhow. However, HOAS forces us to account for them at all times and prevents us from leaving them as free variables. Our solution is to iterate through the list of bindings and substitute $\top$ for each of them. Another possible approach to E-$\Lambda$ is the following two rewrites:

$$\mathit{erase}\{\Lambda\, \overrightarrow{\alpha}.\ e\} \leftarrow [\text{E-}\Lambda] \rightarrow \mathit{erase}\{\Lambda'\, \overrightarrow{\alpha}.\ \mathit{erase}\{e\}\}$$
$$\mathit{erase}\{\Lambda'\, \overrightarrow{\alpha}.\ e[]\} \leftarrow [\text{E-}\Lambda'] \rightarrow e[]$$

In other words, we first $\mathit{erase}\{\}$ the body of the $\Lambda$ term, then discard the variable bindings when there are no references to them. This is a bit more satisfying, since we do not need to perform semantically suspicious substitutions or even treat each binding individually (the vector notation isn't hiding anything in E-$\Lambda'$). However, this approach would require revisiting the $\Lambda$ terms after processing their subterms, which would complicate the tactic code somewhat. We have opted for the simpler solution in our compiler.

### 3.2.5 Type Checking

Type checking, represented by the judgment $\Gamma \vdash e \in T$, is implemented formally, using inference rules. The set of rules, given in figure 3.9, is typical for a specification of an ML-style type system. In the T-TYAPPLY rule we have used the notation $T_1[\overrightarrow{T_2}/\overrightarrow{\alpha}]$ to denote the substitution of the types $\overrightarrow{T_2}$ for the variables $\overrightarrow{\alpha}$ in type $T_1$. Type checking is syntax-directed, so we can once again employ a MetaPRL resource for dispatch.

The type checking rules use two related judgments. We use a *kind* system which currently only contains one kind, $\Omega$. For each primitive type $T$ in the type system we define a well-formedness judgment $\vdash T \in \Omega$. Well-formedness rules for composite types are inductively defined. In addition, we define a type equality judgment $\Gamma \vdash T_1 = T_2$, which tests for structural equality. In our current type system type equality is the same as alpha-equivalence, but we anticipate extending the type system with features that do not maintain this invariant.

The typed AST carries some redundant type information in order to make type checking simpler. For example, in a minimal TAST, an `Apply` term wouldn't need to carry any types at all. However, type checking would require a type environment to supply the types of the function arguments. Alternately, a `Typeof{e}` type could be used, at the expense of computing the type of $e$ multiple times. This is another case where we have chosen a less efficient implementation for the sake of simplicity.

$$\frac{\Gamma; \overrightarrow{v{:}T_1} \vdash e \in T_2}{\Gamma \vdash \lambda \overrightarrow{v{:}T_1}.\ e \in \overrightarrow{T_1} \to T_2}\ \text{T-}\lambda$$

$$\frac{\Gamma \vdash \overrightarrow{e} \in \overrightarrow{T_1} \quad \Gamma \vdash f \in \overrightarrow{T_1} \to T_2}{\Gamma \vdash f(\overrightarrow{e} : \overrightarrow{T_1}) \in T_2}\ \text{T-APPLY}$$

$$\frac{\Gamma \vdash e_1 \in T_1 \quad \Gamma; v{:}T_1 \vdash e_2[v] \in T_2}{\Gamma \vdash \textbf{let}\ v{:}T_1 = e_1\ \textbf{in}\ e_2[v] \in T_2}\ \text{T-LET}$$

$$\frac{\Gamma; \overrightarrow{\alpha} \in \Omega \vdash e \in T}{\Gamma \vdash \Lambda\, \overrightarrow{\alpha}.\ e \in T}\ \text{T-}\Lambda$$

$$\frac{\Gamma \vdash e \in \forall\, \overrightarrow{\alpha}.\ T_1 \quad \Gamma; \vdash T_1[\overrightarrow{T_2}/\overrightarrow{\alpha}] = T_3}{\Gamma \vdash \texttt{TyApply}\{e{:}\forall\, \overrightarrow{\alpha}.\ T_1;\ \overrightarrow{T_2}\} \in T_3}\ \text{T-TYAPPLY}$$

$$\frac{\Gamma \vdash e \in T_1 \quad \Gamma \vdash T_1 = T_2}{\Gamma \vdash \texttt{Constrain}\{e{:}T_1\} \in T_2}\ \text{T-CONSTRAIN}$$

$$\frac{}{\Gamma; v{:}T; \Delta[v] \vdash\, !v \in T}\ \text{T-VAR}$$

Figure 3.9: Type checking rules for the core TAST

It should be noted that there are two known flaws in the type checking rules described here. First, they do not verify the value restriction that was introduced in the type inference section. This means that if an error in the type inference algorithm was to generate a program that violated the value restriction, the type checker would not detect it, and the output of the compiler could be corrupted. This is just the kind of scenario we were trying to avoid by using formal methods! Later versions of the compiler have corrected this oversight by verifying the value restriction during type checking.

An additional flaw is that type well-formedness is not verified in most of the rules. This is a less serious problem, since there are many parts of the compiler that will fail if a type is malformed. It is extremely unlikely that a program containing such a type could be compiled, so this problem would probably only lead to failed compilations, not corrupted ones. Nonetheless, we have corrected the problem in more recent versions of the compiler.

### 3.2.6 CPS Conversion

The implementation of CPS conversion is a good illustration of our methodology. We wish to demonstrate both that 1) the formal definition of the compiler transformations is natural, and 2) that the methodology is compositional. We present a very straightforward implementation based on the ability of the framework to combine the meta-language and the object language.

We use a higher-order variant of Danvy and Filinski's approach to CPS conversion [4]. We start by adding a new term to the meta-language, **let_cps** $v = [\![e : T]\!]$ **in** $c[v]$, where $e$ is the expression

that is being converted, $T$ is the type of that expression and $c$ is the *meta-continuation* of the CPS process. In other words, $c$ is the *rest* of the program and $v$ marks the location where the CPS of $e$ should go. The semantic brackets around $e$ and $T$ signify that these expressions must be CPS-transformed before being substituted into $c$. Note that we use meta-language notation in place of Danvy and Filinski's "static" operators $\overline{@}$ and $\overline{\lambda}$.

The following rule specifies CPS for variables.

$$\textbf{let\_cps}\ v = [\![(!x) : T]\!]\ \textbf{in}\ c[v] \leftarrow [\text{CPS-VAR}] \rightarrow c[!x]$$

In this rule, the meta-continuation is consumed. The rewrite puts the variable into the appropriate location and returns the whole expression.

In the rule for **let** expressions, a new meta-continuation is created.

$$\begin{aligned}
&\textbf{let_cps}\ v_2 = [\![(\textbf{let}\ v_1 : T_1 = e_1\ \textbf{in}\ e_2[v_1]) : T_2]\!]\ \textbf{in}\ c[v_2] \\
&\quad \leftarrow [\text{CPS-LET}] \rightarrow \\
&\textbf{let_cps}\ v_3 = [\![e_1 : T_1]\!]\ \textbf{in} \\
&\textbf{let}\ v_1 : \texttt{TyCPS}\{T_1\} = v_3\ \textbf{in} \\
&\textbf{let_cps}\ v_2 = [\![e_2[v_1] : T_2]\!]\ \textbf{in}\ c[v_2]
\end{aligned}$$

`TyCPS` is a meta-term that is used to specify the CPS conversion for types, in a similar way to how the **let_cps** term is used to specify the CPS conversion for expressions. It is quite simple, since the only modification that needs to be made is the addition of an extra continuation type to function types. Here is the rewrite for that step:

$$\begin{aligned}
&\texttt{TyCPS}\{\overrightarrow{T} \rightarrow T_r\} \\
&\quad \leftarrow [\text{TyCPS-TYFUN}] \rightarrow \\
&((T_r \rightarrow \bot), \texttt{TyCPS}\{\overrightarrow{T}\}) \rightarrow \bot
\end{aligned}$$

The rule for the CPS of applications could be specified the following way:

$$\begin{aligned}
&\textbf{let_cps}\ v = [\![f(\overrightarrow{e} : \overrightarrow{T_e}) : T]\!]\ \textbf{in}\ c[v] \\
&\quad \leftarrow [\text{CPS-APPLY}] \rightarrow \\
&\textbf{let_cps}\ f' = [\![f : \overrightarrow{T_e} \rightarrow T]\!]\ \textbf{in} \\
&\textbf{let_cps}\ v_e = [\![\overrightarrow{e} : \overrightarrow{T_e}]\!]\ \textbf{in} \\
&\textbf{let}\ c' : (\texttt{TyCPS}\{T\} \rightarrow \bot) = \lambda\, v : \texttt{TyCPS}\{T\}.\ c[v]\ \textbf{in} \\
&f'((c', v_e) : ((\texttt{TyCPS}\{T\} \rightarrow \bot), \texttt{TyCPS}\{\overrightarrow{T_e}\}))
\end{aligned}$$

The function and its arguments are CPS-converted, and a new continuation $c_2$ is created that can

take the result of the application and pass it to the original continuation. The application is then rewritten to use pass $c_2$ as the first argument (the continuation argument) of $v_f$.

However, in our implementation we add a meta-let operation (with the usual semantics) to the meta-language.

$$\textbf{meta\_let}\ v = e_1\ \textbf{in}\ e_2[v] \leftarrow [\textsc{meta\_let}] \rightarrow e_2[e_1]$$

Using this operation, the CPS-APPLY rule is written as follows.

$$\begin{array}{l}
\textbf{let_cps}\ v = [\![f(\overrightarrow{e} : \overrightarrow{T_e}) : T]\!]\ \textbf{in}\ c[v] \\
\quad \leftarrow [\text{CPS-APPLY}] \rightarrow \\
\textbf{let_cps}\ f' = [\![f : \overrightarrow{T_e} \rightarrow T]\!]\ \textbf{in} \\
\textbf{let_cps}\ v_e = [\![\overrightarrow{e} : \overrightarrow{T_e}]\!]\ \textbf{in} \\
\textbf{meta_let}\ T' = \texttt{TyCPS}\{T\}\ \textbf{in} \\
\textbf{meta_let}\ T_{c'} = T' \rightarrow \bot\ \textbf{in} \\
\textbf{let}\ c' : T_{c'} = \lambda\, v : T'.\ c[v]\ \textbf{in} \\
f'((c', v_e) : (T_{c'}, \texttt{TyCPS}\{\overrightarrow{T_e}\}))
\end{array}$$

This is more efficient because the reduction tactic for CPS only needs to compute the types once, instead of three times. Again, the ability to combine the object language with meta-language can yield very compact, straightforward, and precise formal code.

The ability to manipulate the meta-continuations also helps make the rules for the conversion of the argument lists very concise. Notice that the list is built from the outside in and does not need to be reversed!

$$\begin{array}{l}
\textbf{let_cps}\ v = [\![() : ()]\!]\ \textbf{in}\ c[v] \\
\quad \leftarrow [\text{CPS-ARGS-NIL}] \rightarrow \\
c[()]
\end{array}$$

$$\begin{array}{l}
\textbf{let_cps}\ v = [\![(e_1 :: \overrightarrow{e}) : (T_1 :: \overrightarrow{T})]\!]\ \textbf{in}\ c[v] \\
\quad \leftarrow [\text{CPS-ARGS-CONS}] \rightarrow \\
\textbf{let_cps}\ v_1 = [\![e_1 : T_1]\!]\ \textbf{in} \\
\textbf{let_cps}\ v_r = [\![\overrightarrow{e} : \overrightarrow{T}]\!]\ \textbf{in} \\
c[(v_1 :: v_r)]
\end{array}$$

In addition to the basic CPS transformation, we use optimized rewrites to prevent CPS from generating a superfluous continuation for function calls in tail positions. When transforming the body of a function definition, we add the extra continuation parameter and use a **tail_cps** term instead of **let_cps**.

$$\begin{array}{l}
\textbf{let_cps}\ v = [\![(\lambda\, \overrightarrow{v : T}.\ e) : (\overrightarrow{T} \rightarrow T_e)]\!]\ \textbf{in}\ c[v] \\
\quad \leftarrow [\text{CPS-}\lambda] \rightarrow \\
c[\lambda\, (c' : (\texttt{TyCPS}\{T_e\} \rightarrow \bot), \overrightarrow{v : \texttt{TyCPS}\{T\}}). \\
\quad \textbf{tail_cps}\ c'([\![e : T_e]\!])]
\end{array}$$

$$
\begin{array}{l}
\textbf{tail\_cps}\ c([\![f(\overrightarrow{e} : \overrightarrow{T_e}) : T]\!]) \\
\quad \leftarrow [\text{CPS-APPLY-TAIL}] \rightarrow \\
\textbf{let\_cps}\ f' = [\![f : \overrightarrow{T_e} \to T]\!]\ \textbf{in} \\
\textbf{let\_cps}\ v_e = [\![\overrightarrow{e} : \overrightarrow{T_e}]\!]\ \textbf{in} \\
\textbf{meta\_let}\ T_c = \texttt{TyCPS}\{T\} \to \bot\ \textbf{in} \\
\textbf{let}\ c' : T_c = c\ \textbf{in} \\
f'((c', v_e) : (T_c, \texttt{TyCPS}\{\overrightarrow{T_e}\}))
\end{array}
$$

$$
\begin{array}{l}
\textbf{tail\_cps}\ c([\![\textbf{let}\ e_1 : T_1 = v\ \textbf{in}\ e_2[v] : T_2]\!]) \\
\quad \leftarrow [\text{CPS-LET-TAIL}] \rightarrow \\
\textbf{let\_cps}\ w = [\![e_1 : T_1]\!]\ \textbf{in} \\
\textbf{let}\ v : \texttt{TyCPS}\{T_1\} = w\ \textbf{in} \\
\textbf{tail\_cps}\ c([\![e_2[v] : T_e]\!])
\end{array}
$$

Figure 3.10: Tail-optimized CPS rewrites

The CPS-Λ rule also uses **tail_cps** to transform its body for the same reason.

The CPS-LET-TAIL rule is a modified version of the normal CPS-LET rule that passes the **tail_cps** through to the tail position, using the normal CPS transformation for the value being bound. If an application is found in tail position, the CPS-APPLY-TAIL rule is used to process it without creating an extra continuation. These rewrites are shown in figure 3.10. Notice that this rule is essentially the same as CPS-APPLY, the only difference being the definition of $c'$. One might wonder why $c'$ is defined at all in CPS-APPLY-TAIL since it's just an alias for $c$ and it only appears once in the rest of the program. The reason is that CPS-APPLY-TAIL is a *derived* rewrite. The let definition is left in to make it easier to formally prove that CPS-APPLY-TAIL can be derived from CPS-APPLY. (The binding itself will be optimized away in a later stage.) In fact, CPS-LET-TAIL is also a derived rewrite, based on the CPS-LET rule. To prove these derivations, however, we also need provide the definition of **tail_cps** in terms of **let_cps**. This is done in the CPS-TAIL rewrite.

$$
\begin{array}{l}
\textbf{tail\_cps}\ c([\![e : T]\!]) \\
\quad \leftarrow [\text{CPS-TAIL}] \rightarrow \\
\textbf{let\_cps}\ v = [\![e : T]\!]\ \textbf{in}\ c(v : \texttt{TyCPS}\{T\})
\end{array}
$$

This definition and an $\eta$-expansion rewrite suffice to prove the derived rewrites. In addition to aiding us in our proofs, CPS-TAIL allows the compiler to fall back to the unoptimized case if there is no match for **tail_cps** $c([\![e : T]\!])$ in the `cps` resource for some specific expression $e$. This will happen, for example, when a function does not end with a tail call or an extension fails to provide a tail-optimized version of its CPS transformation.

### 3.2.7 Closure Conversion

Closure conversion is a stage to which HOAS brings both advantages and disadvantages. The notions of closed terms and free variables are primary in HOAS, which makes much of the closure process trivial to express. On the other hand, HOAS prevents capturing substitution, which is normally an

advantage but can be a hindrance during closure conversion.

In closure conversion we wish to transform every function in the program so that it contains no free variables. This is accomplished in three stages. First, each function definition is transformed to be the application of a *closure abstraction*. The corresponding rewrite is:

$$\lambda\,\overrightarrow{v{:}T}.\ e \leftarrow [\textsc{add\_frame}] \rightarrow (\lambda_c\,().\ \lambda\,\overrightarrow{v{:}T}.\ e)((){:}())_c$$

We define a new lambda form, $\lambda_c\,\overrightarrow{v{:}T}.\ e$, to represent closure abstraction. The corresponding application form, $c(\overrightarrow{e}{:}\overrightarrow{T_e})_c$, represents allocation of the environment and construction of the closure. Initially each function is defined with an empty closure applied to no expressions.

During the second stage, each free variables in the function is re-bound immediately above the closure definition. We introduce a special type of *let* expression for this stage and apply an inverse-beta reduction rewrite:

$$e[!v] \leftarrow [\textsc{beta\_inverse}] \rightarrow \mathbf{let}_c\ x{:}T = !v\ \mathbf{in}\ e[x]$$

Clearly this rewrite cannot be applied in an automated fashion. For any given term it would not be clear which free variable $v$ would refer to. More troubling, semantically speaking, is the type $T$, which does not appear on the left side of the rewrite. The context-free nature of rewrites implies that the rewrite is valid for *any* choice of $T$, which is clearly not true[1]. This is an example of a rewrite that is not generally semantics-preserving. Because of this, we must trust the tactic code that drives the closure conversion stage to perform a type check afterwards.

In order to apply the beta_inverse rewrite, an informal tactic must be written that specifies $!v$ and $T$ by providing the contractum of the rewrite. We must be careful in writing this tactic, however, because the type $T$ that it provides must be correct or the compilation can be corrupted. The tactic itself is straightforward. For each free variable in each closure, the tactic applies beta_inverse with the closure application as $e$. A type environment is used to provide the correct type $T$. Thus, eventually each closure looks like:

$$\begin{aligned}
&\mathbf{let}_c\ x_1{:}T_1 = v_1\ \mathbf{in}\\
&\vdots\\
&\mathbf{let}_c\ x_n{:}T_n = v_n\ \mathbf{in}\\
&(c[x_1 \ldots x_n])((){:}())_c
\end{aligned}$$

The third and final stage is to put each $\mathbf{let}_c$-bound variable into the closure. This is straightfor-

[1]Later versions of the compiler have addressed this weakness by using a *sweep* tactic to formally collect the necessary type information.

| | | |
|---|---|---|
| $e$ += | `AtomUnop`$\{o;\ e\}$ | Unary Operator Atom |
| | `AtomBinop`$\{o;\ e_1;\ e_2\}$ | Binary Operator Atom |
| | `AtomRelop`$\{o;\ e_1;\ e_2\}$ | Relational Operator Atom |
| $o$ ::= | `Unop`[opname]$\{T \to T_r\}$ | Unary Operator Prototype |
| | `Binop`[opname]$\{(T_1, T_2) \to T_r\}$ | Binary Operator |
| | `Relop`[opname]$\{(T_1, T_2) \to T_r\}$ | Relational Operator |

Figure 3.11: Operator extension additions to the Typed AST

ward and can be done with a simple resource-driven rewrite:

$$\begin{aligned}
&\mathbf{let}_c\ x{:}T_x = v_x\ \mathbf{in}\ (\lambda_c\, \overrightarrow{w{:}T}.\ e[x])(\overrightarrow{y}[x] : \overrightarrow{T_y})_c \\
&\quad \leftarrow [\text{ADD_TO_FRAME}] \rightarrow \\
&(\lambda_c\, (w_x{:}T_x, \overrightarrow{w{:}T}).\ e[w_x])((v_x, \overrightarrow{y}[w_x]) : (T_x, \overrightarrow{T_y}))_c
\end{aligned}$$

In order to validate the type introduced in the beta_inverse rewrite, and in order to catch potential programming errors in the other rewrites as well, the output of closure conversion is type checked before being passed to the backend.

### 3.2.8 The x86 Backend

The author was not involved in the design or implementation of the x86 backend. Accounts of this work can be found in [11] and [12].

## 3.3 Compiler Extensions

In Section 3.1 we gave an overview of the compiler's architecture, describing the separation of the compiler core from the extensions used to implement most language features. Here we describe the implementation of several of the extensions to the core compiler.

### 3.3.1 Operators

The operator extension provides the ability to define and use unary, binary, and relational operators. It does not define any specific operators, but provides prototypes that can be specialized by other extensions. This allows the type inference and type checking rules to be implemented once, in the operator extension, and then reused by all other extensions that need to create operators.

The operator extension augments the typed AST as described in figure fig-op-extension. `Relop`s are relational operators such as the integer operations < and <=. They are kept separate from ordinary `Binop`s as a convenience for the backend, where they need to be treated differently from regular binary operators.

Type inference and checking rules are extended to include the new expressions. For example, the following rule is added to type check unary operators.

$$\frac{\Gamma \vdash e \in T_1 \quad \Gamma \vdash t = T_2 \quad \Gamma \vdash \texttt{IsOp\{Unop[n]\{T}_1 \rightarrow \texttt{T}_2\texttt{\}\}}}{\Gamma \vdash \texttt{AtomUnop\{Unop[n]\{T}_1 \rightarrow \texttt{T}_2\texttt{\}; e\}} \in t}$$

The $\Gamma \vdash \texttt{IsOp\{o\}}$ judgment asserts the validity of the operator $o$. The following code to support unary operators is added to the type inference tactic.

```
let infer_unop infer tenv venv s ◁AtomUnop{Unop[op]{T → Tr}; e}▷ =
    let e', s, ty_a = infer tenv venv s e in
    let s = Add the constraint T=ty_a in
        ▷AtomUnop{Unop[op]{T → Tr}; e'}◁, s, ▷Tr◁
```

The code for binary operators and relational operators is analogous. Note that the typed and untyped representations of an operator are the same—ad-hoc polymorphism is not supported. The operator stores the expected types of its operands and its return type, even in the untyped AST. This information is used to automate the type inference and checking rules.

This design makes it extremely simple for an extension to add a new operator. Only two additions are required: an addition to the parser, and an `IsOp{}` rule. In return, type checking and type inference come for free. For example, the integer extension defines the following `IsOp{}` rule for multiplication.

$$\Gamma \vdash \texttt{IsOp\{Binop[“*”]\{}(\mathbb{I}, \mathbb{I}) \rightarrow \mathbb{I}\texttt{\}\}}$$

If an erroneous `Binop` (say $\texttt{Binop[“*”]\{(string}, \mathbb{I}) \rightarrow \mathbb{I}\}$) appeared in the program, perhaps because of a parser error, it would be caught as a type error because there would be no corresponding `IsOp{}` rule.

### 3.3.2 Unit, Booleans, Integers, and String Literals

It is very easy to extend the compiler with new base types. The unit extension, which provides the unit literal and unit type, is probably one of the smaller extensions imaginable, weighing in at just 51 lines of code.

The Boolean extension provides the *true* and *false* literals, **if** / **then** / **else** expressions, and the Boolean type $\mathbb{B}$. Logical operations on booleans are also provided by building on the operator extension.

The integer extension provides integer values, types and operations. We use a 31-bit representation of integers, setting the least-significant bit to 1 in order to distinguish integer values from word-aligned pointers for the garbage collector. The entire extension is about 190 lines of code.

The string extension supports the use of string literals in programs. Useful operations such as

subscripting, concatenation, and copying are not supported, but adding them would be straightforward.

### 3.3.3 Tuples and Arrays

The tuple and array extensions add composite types to the compiler. The tuple extension provides a tuple constructor, tuple subscript operator, and product type. The array extension adds a mutable array type to the language. Operations are provided for creating arrays and getting and setting array elements. In this version of the compiler, arrays and tuples are the only composite types. Later versions also include reference cells and existential types.

When imperative features such as arrays are added to the language, knowing the language's order of operation becomes important for understanding its semantics. The order of operation in the current compiler is not modularly defined. It is established as a side-effect of the rules for the CPS transformation stage. This is an area that future versions of the compiler could improve upon.

### 3.3.4 Recursive Functions

Recursive functions are provided by a fixpoint operator. For this reason the extension is called the fix extension. The source-level syntax for recursive functions is

$$e ::= \textbf{let rec } f(v_1, \cdots, v_n) = e \textbf{ in } e$$

where $f$ is in scope in both subexpressions. In order to give the function's body access to its name, recursive functions are represented as taking an extra parameter. This is the untyped AST representation:

$$e ::= \lambda_{ur}(\overrightarrow{v}, f).\ e[f]$$

The representation is similar in the rest of the compiler, essentially mirroring the account of ordinary functions. Many of the rewrites and inference rules we have described for typed and untyped functions are parameterized over the kind of function, which can be either standard, closure, or recursive. This allows us to reuse any rules that are truly generic and only override those that need to be customized. The only non-standard step for recursive functions is type inference, in which care must be taken to treat the $f$ parameter properly.

```
let infer_lambda_rec infer tenv venv s ◁λur(v→, f). e▷ =
    let a = A new type variable in
    let venv, tenv = Update environments with f:a in
    let f_t, s, ty_f = infer tenv venv s (▷λu(v→). e◁) in
    (match f_t with
        ◁λ(v_t→). e_t▷ ->
            let s = Add the constraint a=ty_f in
                ▷λr(v_t→, f : ty_f). e_t◁, s, ▷ty_f◁)
```

The fix extension is very simple (110 lines of code) but it only provides single-recursion. This is sufficiently general for a proof of concept but not ideal for a production language. In later versions of the compiler we have implemented mutual recursion using a declarations/definitions scheme.

# Chapter 4

# Results

In this thesis we have presented a new methodology for compiler construction along with a case study. The compiler we have created is primitive but complete, in that it can compile simple programs to valid x86 assembly. Our goal was to develop techniques that would enhance reliability and modularity. Our methodology has several desirable properties when compared to traditional techniques:

1. Isolation of trusted code: The trusted core of the compiler is kept small and isolated from the untrusted portion. Although this was not fully realized in early versions of the compiler, the trusted core is normally implemented using formal rules and rewrites, which are generally concise and easy to reason about. Furthermore, the formal framework makes a number of syntactic guarantees about rewrites that help minimize implementation errors. In compilers implemented using traditional general-purpose languages there is normally no such isolation of trust or protection from erroneous transformations.

2. Declarative syntax: Another advantageous property of the trusted core is that it is implemented in a declarative syntax that allows the programmer to express code transformations in a manner that closely mirrors textbook accounts. In many cases the implementation of a transformation is a one-to-one translation of a formal description of its operational semantics. This is clearly not the case with compilers implemented using general-purpose languages.

3. Compositionality: This property is achieved by judicious use of MetaPRL resources for extensible pattern matching. Orthogonal features can be easily be added and removed from the source language. The framework also provides tools for managing non-orthogonal features, including allowing dependencies between extensions. The source code files for extensions are self-contained, allowing anybody reading the code to quickly understand the semantics of each extension. In many traditional compilers, language feature implementations are intertwined throughout the entire code base.

Although it is difficult to draw conclusions based on code size, it is encouraging to note that the size of our compiler is quite small. According to David A. Wheeler's "SLOCCount" tool, our compiler contains about 7900 physical lines of code, with about 4900 of those coming from the x86 backend. This distribution is not surprising since the backend required a much larger proportion of informal code. One large source of informal code was the register allocator, which accounted for about 1600 lines of code. Others were the pretty-printer for the x86 assembly code and various utilities to determine the set of operands and variables used in a given block.

This leaves about 3000 lines of code to implement the front end of the compiler. Previous experience [13] in our group suggests that this is about a factor of 10 fewer lines of code than would be required to implement a similar compiler in OCaml using traditional methodologies. It could be argued that we should include the size of MetaPRL in our calculations, but that would not be appropriate. MetaPRL is a general-purpose logical framework that was not conceived as a compiler-writing toolkit—in fact, its primary design purpose was completely unrelated to compilation.

Our goal in this work was not to produce a verified compiler. There are those who argue that verification is the *only* worthwhile goal in the formalization of programs. While we agree that verification is an important application of formal methods and highly desirable, we disagree that it must be an all-or-nothing affair. Complete formalization places a heavy burden on the programmer, and in many (if not most) cases the benefits of formalization will not outweigh the costs. By pursuing development in the "semi-formal" style that we have outlined in this thesis, the programmer can derive many of the benefits of formal development (improved reliability and declarative specification, for example) without needing to push the formalization into every corner of the compiler.

However, we do not want to close the door on verification, so our techniques have been designed to produce artifacts that are amenable to verification. It is interesting to note that only 236 rewrites were required to implement the compiler. Thanks to the separation of trusted and untrusted code, these rewrites are nearly the only parts of the compiler that would need to be verified if full verification was desired. There are few enough of them that such a task seems feasible. Also, if full verification is not required but an added degree of confidence is needed then partial verification is possible.

# Chapter 5

# Future Work

This thesis describes what was essentially the first working version of our compiler. Since then there have been many improvements made to the compiler. Some of these are:

- Most of the invalid rewrites in the compiler have been eliminated, strengthening the separation between trusted and untrusted code.
- A type system has been added to MetaPRL which adds (meta-)types to the term language. This has helped us to detect and eliminate a large number of bugs.
- MetaPRL theories can now define custom, modular grammars that enable us to specify our rewrites and rules in a syntax very similar to the pretty-printed syntax we use in this thesis. This has made the rules much more concise than their original raw MetaPRL term syntax implementations.
- All functions are now treated as recursive, and true mutual recursion is possible. A list of mutually recursive functions is represented using two nested sequents, with the outer sequent acting as a list of declarations and the inner sequent acting as a list of definitions.
- Several optimizations have been implemented, including direct-call optimization, inlining, and dead-code elimination.
- Several new imperative extensions have been added to the compiler, including reference cells, sequencing, call/cc, and loops.

There is still work to be done, however. We would like to explore more challenging code movement optimizations such as partial redundancy elimination [22]. Formalizing code motion is difficult in HOAS, particularly in the presence of imperative features.

Our research group has explored the benefits of speculative execution to simplify programming for distributed systems. As part of this work we would like to add *speculations* [31] to our compiler

as a language primitive. This will require some runtime support, but should be quite simple because speculations are very simple, semantically.

Finally, now that we have established techniques for writing a compiler in a formal toolkit we would like to start proving properties of the compiler, preferably with a high degree of automation. We have begun to explore reflection as a means for meta-reasoning about formal artifacts [26]. We expect that reflection will provide a generic mechanism for automatically internalizing the artifacts specified in a prover.

# Chapter 6

# Related Work

`FreshML` [29] adds to the ML language support for straightforward encoding of variable bindings and alpha-equivalence classes. Our approach differs in several important ways. Substitution and testing for free occurrences of variables are explicit operations in `FreshML`, while MetaPRL provides a convenient implicit syntax for these operations. Binding names in `FreshML` are inaccessible, while only the formal parts of MetaPRL are prohibited from accessing the names. Informal portions—such as code to print debugging messages to the compiler writer, or warning and error messages to the compiler user—can access the binding names, which aids development and debugging. `FreshML` is primarily an effort to add automation; it does not address the issue of validation directly.

Liang [21] implemented a compiler for a simple imperative language using a higher-order abstract syntax implementation in λProlog. Liang's approach includes several of the phases we describe here, including parsing, CPS conversion, and code generation using a instruction set defined using higher-abstract syntax (although in Liang's case, registers are referred to indirectly through a meta-level store, and we represent registers directly as variables). Liang does not address the issue of validation in this work, and the primary role of λProlog is to simplify the compiler implementation. In contrast to our approach, in Liang's work the entire compiler was implemented in λProlog, even the parts of the compiler where implementation in a more traditional language might have been more convenient (such as register allocation code).

Lerner *et. al.* have implemented a framework for compiler experimentation [17, 18] where new compiler optimizations are proven correct automatically. In their method the programming language must stay fixed, so their approach, while great for experimenting with compiler optimizations, is not appropriate for programming language experimentation.

Hannan and Pfenning [8] constructed a verified compiler in LF (as realized in the Elf programming language) for the untyped lambda calculus and a variant of the CAM [2] runtime. This work formalizes both compiler transformation and verifications as deductive systems, and verification is against an operational semantics.

Previous work has also focused on augmenting compilers with formal tools. Instead of trying

to split the compiler into a formal part and a heuristic part, one can attempt to treat the *whole* compiler as a heuristic adding some external code that would watch over what the compiler is doing and try to establish the equivalence of the intermediate and final results. For example, the work of Necula and Lee [23, 24] has led to effective mechanisms for certifying the output of compilers (e.g., with respect to type and memory-access safety), and for verifying that intermediate transformations on the code preserve its semantics. Pnueli, Siegel, and Singerman [30] perform verification in a similar way, not by validating the compiler, but by validating the result of a transformation using simulation-based reasoning.

Semantics-directed compilation [16] is aimed at allowing language designers to generate compilers from high-level semantic specifications. Although it has some overlap with our work, it does not address the issue of trust in the compiler. No proof is generated to accompany the compiler, and the compiler generator must be trusted if the generated compiler is to be trusted.

Boyle, Resler, and Winter [1], outline an approach to building trusted compilers that is similar to our own. Like us, they propose using rewrites to transform code during compilation. Winter develops this further in the HATS system [32] with a special-purpose transformation grammar. An advantage of this approach is that the transformation language can be tailored for the compilation process. However, this significantly restricts the generality of the approach, and limits re-use of existing methods and tools.

An example of a systematic but informal approach to programming language exploration can be found in the interpreters that accompany Pierce's book [28]. Written in OCaml, these interpreters provide implementations of assorted lambda calculi presented in the book. They are very useful for language experimentation, but suffer from various shortcomings typical of tools created using traditional techniques. Features are not compositional, implementation transparency is poor, and there is no hope of reasoning about the implementation.

# Bibliography

[1] J. Boyle, R. Resler, and K. Winter. Do you trust your compiler? Applying formal methods to constructing high-assurance compilers. In *High-Assurance Systems Engineering Workshop*, Washington, DC, August 1997.

[2] G. Cousineau, P.L. Curien, and M. Mauny. The categorical abstract machine. *The Science of Programming*, 8(2):173–202, 1987.

[3] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[4] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[5] The GNU compiler collection. http://gcc.gnu.org/.

[6] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, NY, 1979.

[7] Adam Granicz and Jason Hickey. Phobos: A front-end approach to extensible compilers. In *$36^{th}$ Hawaii International Conference on System Sciences*. IEEE, 2002.

[8] John Hannan and Frank Pfenning. Compiler verification in LF. In *Proceedings of the $7^{th}$ Symposium on Logic in Computer Science*. IEEE, IEEE Computer Society Press, 1992.

[9] Robert Harper, D.B. MacQueen, and R. Milner. Standard ML. Technical Report TR ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, University of Edinburgh, 1986.

[10] Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. In David Basin and Burkhart Wolff, editors, *Proceedings of the $16^{th}$ International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2003.

[11] Jason Hickey, Aleksey Nogin, Adam Granicz, and Brian Aydemir. Compiler implementation in a formal logical framework. In *Proceedings of the 2003 workshop on Mechanized reasoning about languages with variable binding*, pages 1–13. ACM Press, 2003. `http://doi.acm.org/10.1145/976571.976575`. Extended version of the paper is available as Caltech Technical Report caltechCSTR:2003.002.

[12] Jason Hickey, Aleksey Nogin, and Nathaniel Gray. Programming language experimentation using proof assistants. To be submitted to the Journal of Functional Programming, 2005.

[13] Jason Hickey, Justin D. Smith, Brian Aydemir, Nathaniel Gray, Adam Granicz, and Cristian Ţăpuş. Process migration and transactions using a novel intermediate language. Technical Report caltechCSTR:2002.007, California Institute of Technology, Computer Science, August 2002.

[14] Jason J. Hickey et al. Mojave research project home page. `http://mojave.caltech.edu/`.

[15] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[16] Peter Lee. *Realistic compiler generation*. MIT Press, 1989.

[17] Sorin Lerner, Todd D. Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 220–231. ACM, 2003.

[18] Sorin Lerner, Todd D. Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 364–377. ACM, 2005.

[19] Xavier Leroy. *The Objective Caml System: Documentation and User's Manual*, 2002. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Availiable from `http://www.ocaml.org/`.

[20] Michael E. Lesk. LEX — A lexical analyzer generator. Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[21] Chuck C. Liang. Compiler construction in higher order logic programming. In *Practical Aspects of Declarative Languages*, volume 2257 of *Lecture Notes in Computer Science*, pages 47–63, 2002.

[22] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

[23] George C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.

[24] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, 1998.

[25] Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. In Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors, *Proceedings of the 15$^{th}$ International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, volume 2410 of *Lecture Notes in Computer Science*, pages 281–297. Springer-Verlag, 2002.

[26] Aleksey Nogin, Alexei Kopylov, Xin Yu, and Jason Hickey. A syntactic approach to computational reflection. Submitted to TPHOLs 2005.

[27] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, volume 23(7) of *SIGPLAN Notices*, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

[28] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[29] Andrew M. Pitts and Murdoch Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.

[30] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Lecture Notes in Computer Science*, 1384:151–166, 1998.

[31] Cristian Ţăpuş, Justin D. Smith, and Jason Hickey. Kernel level speculative DSM. In *IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2003)*, Tokyo, Japan, 2003.

[32] Victor L. Winter. Program transformation in HATS. In *Proceedings of the Software Transformation Systems Workshop*, May 1999.