

Building Extensible Compilers in a Formal Framework^{*}

A Formal Framework User’s Perspective

Nathaniel Gray, Jason Hickey, Aleksey Nogin, and Cristian Tăpuș

Caltech, M/C 256-80
1200 E California Blvd
Pasadena, CA 91125, USA
{n8gray,jyh,nogin,crt}@cs.caltech.edu

Abstract. We outline a new methodology for compiler design, based on the use of a transformation logic defined within an existing general-purpose logical framework. We discuss how this methodology can be used to address several central issues in compiler design and implementation: ease of implementation, extensibility, compositionality, and trust. We show how pre-existing features of the logical framework we use help in compiler implementation; and we also discuss which features need to be added to the framework in order to facilitate our approach to compiler development.

1 Introduction

We are developing a new methodology for compiler design, based on the use of a transformation logic defined within an existing general-purpose logical framework. In our approach the central part of the compiler is a set of specifications on a formal language; these specifications follow a standard textbook account of programming language semantics almost to the letter. Most of the work required to turn these specifications into an actual compiler is handled *automatically* by the logical framework. We demonstrate how this methodology can be used to address several central issues in compiler design and implementation: ease of implementation, extensibility, compositionality, and trust.

We use the **MetaPRL** formal tool [9,11], which provides a well-defined syntax of terms, types, and programs. We represent programs and program transformations using higher-order abstract syntax (HOAS); binding, scoping, and substitution are handled automatically by the framework. The HOAS also allows mixing the object language (that contains operators like “**let**”) with the meta-language (that contains operators like “**CPS**”), explicitly expressing the intermediate states of the compilation process. In addition, the framework provides a

^{*} This work was supported in part by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research (ONR) under Grant N00014-01-1-0765, the Defense Advanced Research Projects Agency (DARPA), the United States Air Force, the Lee Center, and by NSF Grant CCR 0204193.

rich tactic language for guiding proofs and transformations and for automatically extracting such guidance information from annotated specifications. Finally, the framework provides us with an interactive program refinement mode (initially designed for interactive formal proof development) and together with the explicit meta-language it proved to be an extremely powerful debugging tool.

Compositionality is a well-established principle in the construction of logical theories. In the compiler domain, we take a similar approach to compositionality and extensibility. The compiler defines a core theory for System F (variables, functions, application, and second order quantifiers) that is divided into transformation stages including type inference, type checking, CPS transformation, closure conversion, and assembly code generation. Additional components for Boolean values, arithmetic, tuples, arrays, recursive functions, etc., are defined as independent extensions. Each extension defines its own set of formal rules for each transformation stage and adds new strategy code to the tactic used to control that stage. By locally ensuring that the component acts as a conservative extension of the core and other components it is derived from, we get a strong guarantee that there will be no unexpected interactions between different compiler modules or different language features.

Another extremely important and challenging issue in compiler development is reliability and trust. In the context of a compiler, it is useful to focus on the code where flaws have the potential to cause the compiler to produce incorrect output for some input program—we call such code *trusted*. Flaws in *untrusted* code may cause the compiler to fail to produce output on some valid input programs, but they cannot cause the compiler to produce incorrect output.

When a compiler is implemented in a general purpose language, it is often difficult to isolate the parts of the compiler that must be trusted, and in the worst case the entire code base must be trusted. Trust is also a central issue in compositionality and ease of implementation. If the invariants that specify the compiler involve complex interactions between many parts of the implementation, maintaining and extending the compiler can be quite difficult.

In our approach, the compiler is built in the style of the LCF theorem prover [4]. The program transformations are each defined in two parts: a set of trusted transformation *axioms* and untrusted *tactic code* to direct the transformation strategy. The transformation axioms are defined in a formal logic using notation similar to that in the literature, they represent only a small part of the compiler, and they are *verifiable*. That is, the entire trusted code path is small, precisely and formally defined, and it may be validated against a program semantics if desired. Note, however, that we do not consider verifiability to be the primary concern of this work. We believe that there is substantial value in significantly reducing the amount of trusted compiler code, even if it is not completely eliminated.

A number of guarantees are provided by the framework itself. For example, the HOAS implementation ensures that program transformations are never allowed to violate scoping or accidentally capture a variable. Even the framework implementation does not have to be trusted—the tool is capable of retaining

and providing a full log of the program transformations performed during the compilation process; if an extreme level of confidence is needed, an independent checker could be implemented.

1.1 Overview

This paper is based on a case study of a working compiler implementation for an ML-like source language [7], compiled to assembly code for the Intel x86 machine architecture. As mentioned, the core is based on the language of System F. There are extensions for 1) additional base types like Boolean values and integers, 2) aggregates like arrays and tuples, and 3) recursive functions. The backend uses HOAS to define a scoped x86 assembly language [7,10]. The compiler stages include type inference, type checking, CPS transformation, closure conversion, and assembly code generation. The compiler is implemented in the MetaPRL logical framework.

This paper focuses on demonstrating how the features of the logical framework help to implement the compiler and improve its trustworthiness and extensibility. In our implementation we were able to precisely and concisely define each of the standard compiler stages (excluding parsing and pretty-printing of the output assembly) formally. The precision comes from using the formal notation, and the brevity follows from the rich set of tools provided by the logical framework. We begin the account with a description of terminology (Section 2) and the overall compiler architecture (Section 3), and follow it with a description of a few of the key stages of the compiler. As a demonstration of our approach, we present the CPS stage of the compiler (Section 4) based on the work of Danvy and Filinski [3] and show how the use of HOAS and derived rules in logical framework can make our *implementation* simpler than Danvy and Filinski’s original account. Finally, Section 5 provides a discussion of our experiences and give some ideas for further improvements of the methodology and Section 6 discusses related work.

2 MetaPRL

All logical syntax in the MetaPRL framework is expressed in the language of *terms*. The general syntax of all terms has three parts. Each term has 1) an operator-name (like “sum”), which is a unique name identifying the kind of term; 2) a list of parameters representing constant values; and 3) a set of subterms with possible variable bindings. We use the following syntax to describe terms:

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

All the free occurrences of variables \vec{v}_i in t_i will be considered bound by the operator. When $n = 0$, the parameter brackets are omitted; when \vec{v}_i is empty, the dot before t_i is usually omitted.

Below are a few examples of terms that could be used in a formalization of a simple lambda calculus.

Pretty-printed form	Term
1	<code>integer[1]{}</code>
$\lambda x.b$	<code>lambda[] { x. b }</code>
$f(a)$	<code>apply[] { f; a }</code>
$x + y$	<code>sum[] { x; y }</code>

Numbers have an integer parameter. The `lambda` term contains a binding occurrence: the variable x is bound in the subterm b .

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. (More specifically, if two operators have different arities, they will be considered to be distinct even if they happen to have the same opname.)

In addition to the basic term language described above, the framework also provides three special kinds of terms. The first one is the simple first-order (object language) variables. These are the variables that can be bound in a term.

Another class of special terms are second-order (meta-level) variables, which are patterns used to define scoping and substitution [16]. A second-order variable pattern has the form $V[v_1; \dots; v_n]$, which represents an arbitrary term that may have free first-order variables v_1, \dots, v_n . The corresponding substitution has the form $V[t_1; \dots; t_n]$, which specifies the simultaneous, capture-avoiding substitution of terms t_1, \dots, t_n for v_1, \dots, v_n in the term matched by V . Second-order variables are used to specify logical rules and term rewrites.

A term rewrite states that 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, β -reduction could be specified with the following rewrite.

$$(\lambda x.v_1[x]) v_2 \leftarrow [\text{beta}] \rightarrow v_1[v_2]$$

The $v_1[x]$ in the redex stands for an arbitrary term that may have free occurrences of the first-order variable x , and v_2 is another arbitrary term. The meta-term $v_1[v_2]$ in the contractum specifies the substitution of the term matched by v_2 for x in v_1 .

Second-order notation can also express the *lack* of bound occurrences of a certain variable. The following rewrite is valid in second-order notation and would be provable in the presence of the β -reduction rewrite.

$$(\lambda x.v[]) 1 \leftarrow [\text{const}] \rightarrow (\lambda x.v[]) 2$$

In the context λx , the second-order variable $v[]$ matches only those terms that do not have x as a free variable. No substitution is performed; the β -reduction of both sides of the rewrite yields $v[] \longleftrightarrow v[]$, which is valid reflexively. Normally, when a second-order variable $v[]$ has an empty argument list `[]`, we omit the brackets and use the simpler notation v .

The last class of special terms is sequents (sometime also called telescope terms) of the form

$$x_1 : t_1; \dots; x_n : t_n \vdash_a c,$$

where n can be 0. The term c is the *conclusion* of the sequent; the terms t_i are its *hypotheses*; the variables x_i introduce binding occurrences (each x_i is bound in all t_j for $j > i$ and in c). Finally, the term a is the *sequent argument* that specifies what kind of sequent it is—essentially the argument plays the same role for sequents as the operator name plays for ordinary terms. Sequent *schemas* [16] may also include *context* meta-level variables that stand for arbitrary lists of hypotheses. For example, the sequent schema

$$\Gamma; x : T[]; \Delta[x] \vdash_{a[]} c[x]$$

(where Γ and Δ are context variables and T , a and c are second-order variables) stands for an arbitrary sequent with at least one hypothesis.

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 Γ . 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. The compilation task is a process of rewriting the source program e to an equivalent assembly program e' .

MetaPRL uses OCaml [19] as its tactic construction language in the LCF style. When an inference rule or a rewrite rule is defined in **MetaPRL**, the framework creates an OCaml expression that can be used to apply the rule. Code to guide the application of rules and rewrites is written in OCaml, using a rich set of primitives provided by **MetaPRL**. In addition, **MetaPRL** automates the construction of most guidance code.

3 Compiler Overview

A compiler is defined by a sequence of transformations that take a program in a source language and translate it to a program in a target language. In this case study, the full source language is an ML-like source language with type inference and higher-order functions and the target language is the x86 assembly language.

Figure 1 shows a diagram of the compiler architecture, where the core and the extensions are represented horizontally. Extensions do not have to define code for each of the stages; for example, closure conversion applies only to functions, and the other extensions may ignore it. Extensions may also have dependencies upon one another, as shown by the arrows on the left of each extension: tuples require integers, which require general operations for arithmetic, which require Boolean values for relations.

The compiler includes an initial informal phase that uses the **Phobos** extensible parser to convert the textual source code to the term representation used by the logical framework [5].

The syntax for the typed intermediate language for the case study is shown in Figure 2. The source language is similar, except it is untyped. For clarity, the

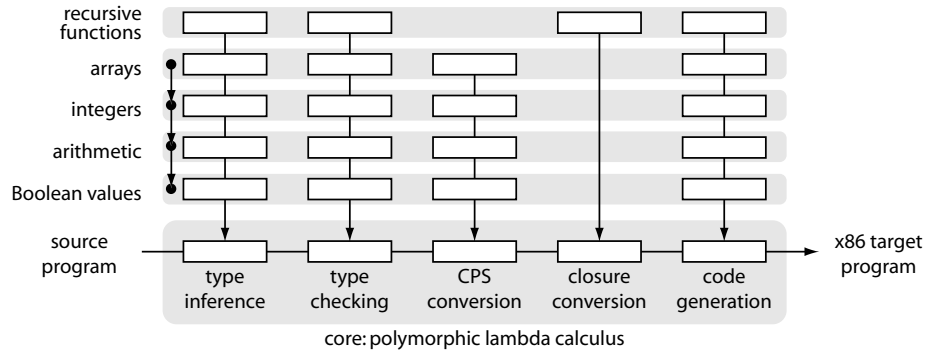


Fig. 1. The high-level compiler architecture is designed around a sequence of transformations for a core language based on the polymorphic lambda calculus. Each extension defines new types and values, as well as an extension to each of the core stages. The vertical arrows indicate extensions to core stages; the code is structured horizontally.

syntax is shown in the pretty-printed form; internally each of the expressions and types uses native MetaPRL notation.

The arities of functions, application, type abstractions, type applications, and tuples are unconstrained. Internally, functions and their types use sequent notation. For example, the sequent $x_1 : t_1, \dots, x_n : t_n \vdash_{\kappa} e$ is used to represent the function $\lambda_{\kappa}(x_1 : t_1, \dots, x_n : t_n).e$. There are three kinds of functions and application: λ_r represents a recursive function (f is the recursive binding); λ_s represents a “normal” function; an application $e(e_1, \dots, e_n : t_1, \dots, t_n)_c$ represents a closure (the runtime passes the arguments as a tuple).

4 Example: 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 and we will show how the tail recursive optimizations can be derived formally from the eta reduction.

We use a higher-order variant of Danvy and Filinski’s approach to CPS conversion [3]. We start by adding a new term to the meta-language— $\text{CPS}\{e; t; v.c[v]\}$, where the first argument e is the expression that is being converted, the second argument t is the type of that expression and the third argument 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 following rule specifies CPS for variables of the object language.

$$\text{CPS}\{!x; t; v.c[v]\} \leftarrow [\text{cps_var}] \rightarrow c[!x]$$

Expressions		Types	
Core language			
$e ::= x$	Variables	$t ::= \alpha$	Variables
$(e : t)$	Type constraint	\perp	Empty type
$\mathbf{let } v : t = e_1 \mathbf{ in } e_2$		\top	All programs
$\lambda_{\kappa}(x_1 : t_1, \dots, x_n : t_n).e$	Functions	$(t_1, \dots, t_n) \rightarrow_{\kappa} t$	Function types
$e(e_1, \dots, e_n : t_1, \dots, t_n)_{\kappa}$	Application		
$\Lambda(\alpha_1, \dots, \alpha_n).t$	Type abstraction	$\forall(\alpha_1, \dots, \alpha_n).t$	Polymorphism
$e[t_1, \dots, t_n]$	Type application		
Boolean values			
$true \mid false$	Constants	\mathbb{B}	Boolean type
$\mathbf{if } e \mathbf{ then } e \mathbf{ else } e$	Conditional		
Integers			
i	Constants	\mathbb{Z}	Integer type
$e \mathit{binop} e$	Arithmetic		
$e \mathit{relop} e$	Relations		
Tuples			
(e_1, \dots, e_n)	Tuples	$t_1 * \dots * t_n$	Product type
$(e : t).i$	Projection		
Recursive functions		Function kinds	
$\lambda_r(x_1 : t_1, \dots, x_n : t_n, f : t).e$		$\kappa ::= s \mid c \mid r$	
	$\mathit{binop} ::= + \mid - \mid \dots$		Binary operations
	$\mathit{relop} ::= < \mid \leq \mid \dots$		Binary relations

Fig. 2. The typed intermediate language is based on the polymorphic lambda calculus. Extensions add Boolean values, arithmetic, tuples, arrays (not shown), and recursive functions. The source language is a type erased version of the intermediate language.

The notation $!x$ is MetaPRL syntax for first-order variables that are bound outside of the local scope of the rewrite rule. In this rule, the meta-continuation is consumed. The rewrite puts the variable into the appropriate location and returns the whole expression. Note that we use meta-language notation in place of Danvy and Filinski’s “static” operators $\bar{\textcircled{a}}$ and $\bar{\lambda}$.

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

$$\begin{aligned}
 & \text{CPS}\{\mathbf{let } v_1 : t_1 = e_1 \mathbf{ in } e_2[v_1]; t_2; v_2.c[v_2]\} \\
 & \leftarrow [\text{cps_let}] \rightarrow \\
 & \text{CPS}\{e_1; t_1; v_3.\mathbf{let } v_1 : \text{TyCPS}\{t_1\} = v_3 \mathbf{ in} \\
 & \quad \text{CPS}\{e_2[v_1]; t_2; v_2.c[v_2]\}\}
 \end{aligned}$$

`TyCPS` here is a meta-term that is used to specify the CPS conversion for types (adding an extra argument to all function types) similarly to how the `CPS` term is used to specify the CPS conversion for expressions.

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

$$\begin{aligned} & \text{CPS}\{f(es : ts); t; v.c[v]\} \\ & \leftarrow [\text{cps_apply}] \rightarrow \\ & \text{CPS}\{f; ts \rightarrow t; v_f. \\ & \text{CPS}\{es; ts; v_e. \\ & \text{let } c_2 : (\text{TyCPS}\{t\} \rightarrow \perp) = \lambda_s v : \text{TyCPS}\{t\}.c[v] \text{ in} \\ & v_f(c_2, v_e : (\text{TyCPS}\{t\} \rightarrow \perp), \text{TyCPS}\{ts\})\} \end{aligned}$$

where es and ts are second-order variables used to match *lists* of arguments and types respectively.

In our implementation we add a meta-let operation to the meta-language.

$$\text{meta.let } v = e_1 \text{ in } e_2[v] \leftarrow [\text{meta.let}] \rightarrow e_2[e_1]$$

Using this operation, the `cps_apply` rule is written as follows.

$$\begin{aligned} & \text{CPS}\{f(es : ts); t; v.c[v]\} \\ & \leftarrow [\text{cps_apply}] \rightarrow \\ & \text{CPS}\{f; ts \rightarrow t; v_f. \\ & \text{CPS}\{es; ts; v_e. \\ & \text{meta.let } t' = \text{TyCPS}\{t\} \text{ in} \\ & \text{meta.let } t'' = t' \rightarrow \perp \text{ in} \\ & \text{let } c_2 : t'' = \lambda_s v : t'.c[v] \text{ in} \\ & v_f(c_2, v_e : t'', \text{TyCPS}\{ts\})\} \end{aligned}$$

This is more efficient as the type t will only have to be converted once, not 3 times. Again, the ability to combine the object language with meta-language yields very compact straightforward and precise formal code.

The ability to manipulate the meta-continuations also helps making the rules for the conversion of the argument lists very concise.

$$\begin{aligned} & \text{CPS}\{e_1 :: es; t_1 :: ts; v.c[v]\} \\ & \leftarrow [\text{cps_args_cons}] \rightarrow \\ & \text{CPS}\{e_1; t_1; v_1.\text{CPS}\{es; ts; v_s.c[v_1 :: vs]\}\} \\ \\ & \text{CPS}\{(); (); v.c[v]\} \leftarrow [\text{cps_args_nil}] \rightarrow c[()] \end{aligned}$$

Below is an example of a CPS rewrite from the Boolean extension, written in the MetaPRL native syntax.

```
prim_rw cps_true { | cps | } :
  CPS{bTrue; TyBool; v. 'c['v]}
  <-->
  'c[bTrue]
```

The above 4 lines are the only code that needs to be added to the system for it to know how to handle the `true` constant in the CPS stage. The system does not require this code to go in a specific place. The `{| cps |}` annotation specifies that this rewrite should be added to the lookup table [8] used by the CPS tactic.

In addition to the basic CPS transformation, we define a tail-recursive version as $\text{TailCPS}\{e; t; k\} := \text{CPS}\{e; t; v.k(v)\}$. Using this definition we *formally derive* the tail call optimizations using the eta reduction rule.

5 Conclusions and Future Work

During the course of this work on the case study, we found that the implementation was easier than we expected, in part because the ability to mix the object and meta-language freely gave us more power than we anticipated. Because the account mirrors standard semantics textbook specifications very closely and the amount of code that must be trusted is only a few hundred lines, it is relatively easy to believe in its correctness. The mechanisms for extensions and compositionality provided by the logical framework generalized naturally to the compiler design.

On the compiler structure side, there are many open avenues to explore. We plan to investigate bounded polymorphism, which we will use for object systems and extensible tuples. The current core language already provides preliminary, but incomplete support. We also plan to develop a representation of mutually recursive functions, which will require extending the support provided by the logical framework.

One apparent challenge of our approach is that all program transformations must be constructed from a fixed number of rewrite rules that each describe a pattern over a fixed number of program points. In other words, global program transformations must be composed of a sequence of local transformations, and it is not always obvious how to do this. In addition, global transformations may require knowledge of the entire program syntax, which can be at odds with compositionality. In our experience, however, we have found this problem to be much easier to solve than we originally expected; all of the transformations we have implemented so far have been easy to break into appropriately localized pieces. On the other hand, we have not yet tried formalizing optimization techniques that are normally implemented using global program analysis, such as global code motion; the problem of breaking these types of transformations into localized rewrites could be harder.

For the most part, our work concentrated on implementing the compiler without modifying the existing logical framework. However in the future we are likely to try adding some additional features to the framework to facilitate compiler implementation. There are two main limitations that we are planning to address—recursive variable-arity binding structure and context-aware rewriting.

5.1 Recursive Binding Structure

Recursive functions are a very basic feature of ML-like languages. In general, recursive functions have the following form.

$$\begin{array}{l} \mathbf{let\ rec}\ f_1\ x_1 \dots x_{k_1} = e_1 \\ \quad \mathbf{and}\ f_2\ x_1 \dots x_{k_2} = e_2 \\ \quad \quad \vdots \\ \quad \mathbf{and}\ f_n\ x_1 \dots x_{k_n} = e_n \\ \mathbf{in}\ e \end{array}$$

There are two difficulties associated with the above—first, the functions have variable arity, and second, the functions are mutually recursive and each of the e_i may have free occurrences of each of the f_j .

As we describe in Section 3, variable arity functions could be implemented by using a sequent representation. Mutual recursion is more challenging. One approach would be to pack the mutually recursive functions into a record and then define the record recursively [10]. Defining a *single* variable recursively is easy, but in this approach function variables turn into explicit record field *names* and are no longer mapped to normal variables. As a result, most of the advantages provided by HOAS are lost and the labels have to be managed (and alpha-renamed) explicitly.

A proper HOAS solution would be to introduce a new kind of sequent to the logical framework—a *recursive sequent* of the form

$$x_1 : t_1 = e_1; \dots; x_n : t_n = e_n \vdash e$$

where each x_i is bound in all the subsequent t_j ($j > i$), in *all* of the e_k ($1 \leq k \leq n$), and in e . The traditional sequent mechanism can be subsumed by recursive sequents by making the e_i optional.

5.2 Context-Aware and Conditional Rewriting

Consider the following trivial optimization rewrite:

$$\mathbf{let}\ v : t = e\ \mathbf{in}\ v \leftarrow [\mathbf{let_opt}] \rightarrow e$$

Depending on the exact semantics used, this rewrite could be considered invalid since it potentially allows turning mistyped expressions into well-typed ones and vice-versa (remember that rewrites are bidirectional). In this simple example, the rewrite can be fixed relatively easily by adding an explicit type constraint to the contractum as follows.

$$\mathbf{let}\ v : t = e\ \mathbf{in}\ v \leftarrow [\mathbf{let_opt}] \rightarrow e : t$$

However, we would generally like to be able to express rewrites that are only conditionally applicable. In particular, we would like to specify conditions of the forms “*applicable in a context that expects the redex to have type t* ” and “*applicable when subterm e is well-typed.*” While the MetaPRL system does provide support for conditional rewriting, not all conditions that are natural in the compiler implementation domain are easily expressible in MetaPRL.

6 Related Work

FreshML [17] 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 [13] implemented a compiler for a simple imperative language using a higher-order abstract syntax implementation in λ Prolog. Liang’s approach includes several of the phases we describe here, including parsing, CPS conversion, and code generation using an 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 λ Prolog is to simplify the compiler implementation. In contrast to our approach, in Liang’s work the entire compiler was implemented in λ Prolog, even the parts of the compiler where implementation in a more traditional language might have been more convenient (such as register allocation code).

Hannan and Pfenning [6] 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 [14,15] 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 [18] 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 [12] 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 [20] 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.

References

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. Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.
4. 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.
5. Adam Granicz and Jason Hickey. Phobos: A front-end approach to extensible compilers. In *36th Hawaii International Conference on System Sciences*. IEEE, 2002.
6. John Hannan and Frank Pfenning. Compiler verification in LF. In *Proceedings of the 7th Symposium on Logic in Computer Science*. IEEE, IEEE Computer Society Press, 1992.
7. Jason Hickey, Nathan Gray, Aleksey Nogin, and Cristian Tapus. Reliable frameworks for extensible compilers. In preparation, 2004.
8. Jason Hickey and Aleksey Nogin. Extensible hierarchical tactic construction in a logical framework. Accepted to the TPHOLs 2004 conference, 2004.
9. 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 16th 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.
10. 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.
11. Jason J. Hickey, Aleksey Nogin, Alexei Kopylov, et al. MetaPRL home page. <http://metapr1.org/>.
12. Peter Lee. *Realistic compiler generation*. MIT Press, 1989.
13. 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.
14. George C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.

15. 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.
16. Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. In Victor A. Carreño, César A. Muñoz, and Sophiène Tahar, editors, *Proceedings of the 15th 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.
17. 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.
18. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Lecture Notes in Computer Science*, 1384:151–166, 1998.
19. Pierre Weis and Xavier Leroy. *Le langage Caml*. Dunod, Paris, 2nd edition, 1999. In French.
20. Victor L. Winter. Program transformation in hats. In *Proceedings of the Software Transformation Systems Workshop*, May 1999.