

Building Reliable Compilers with a Formal Methods Framework

Nathaniel Gray, Cristian Țăpuș, Aleksey Nogin, and Jason Hickey
Caltech Computer Science 256-80, Pasadena, CA 91125
n8gray, crt, nogin, jyh@cs.caltech.edu

1. Introduction

Reliability is an issue in compiler writing just like it is in any other type of large software endeavor. Compiler reliability is particularly important, however, because compilers are the foundation of the software infrastructure – almost every program in use today is compiled. Bugs in compilers are particularly difficult to detect. Even when the compiler is transforming a correct program into incorrect assembly code, the programmer generally trusts the compiler and assumes that his code is faulty.

In principle it should be easy to write reliable compilers. Succinct, unambiguous mathematical formalisms have been created to specify the semantics of programming languages and compilers. The compiler would be much more likely to be correct if it were written in a high-level mathematical language. But in practice, the use of “compiler compilers” has been restricted to the lexing and parsing phases of compilation. The vast majority of compiler code is written in general-purpose programming languages like C or ML that don’t support high-level mathematical reasoning. As a consequence, it is very difficult to judge whether or not the code implements the compiler’s specification correctly.

We have learned this lesson from our own experience. Our first approach to building a reliable compiler was to design a small, well-specified formal intermediate representation (FIR) and then build a traditional compiler that could compile multiple source languages to this representation [3]. Because we had a formal specification for the FIR we were able to reason about its semantics mathematically. However, the compiler itself was written in ML, which made it necessary to translate the specification to verbose ML code. Consequently, the ML translation was much larger than the specification, and errors in the translation were common even though the model itself was sound.

In addition, because of the size of the code (about 300,000 lines), experimenting with new features was prohibitively complex. Adding a new feature to the compiler meant changing a large number of files.

In this paper we describe an ongoing research effort to address these problems by building a compiler within the

MetaPRL formal programming environment [1]. Working with MetaPRL allows us to write the code for our compiler using the same conceptual model that we would use to specify the semantics of a source language or compiler stage. This greatly reduces the amount of trusted code in the compiler, and makes it easier to judge whether or not that code is correct.

We will begin by briefly describing the MetaPRL system. We will then describe the process of writing a compiler using the system. Finally, we will discuss our experiences practicing this methodology and present future directions.

2. Writing a Compiler with MetaPRL

The MetaPRL system is a general-purpose formal tool that combines the properties of an interactive LCF-style tactic-based proof assistant, a logical framework, a logical programming environment, and a formal methods programming toolkit. It is implemented as an extension to the OCaml language and its modular design permits incremental definition of logics.

Broadly speaking, compiler stages can be divided into two categories: transformations and judgments. An example of a transformation is the translation of the program from the source language to an intermediate representation. An important judgment stage is type checking, which does not change any code but verifies that it is well-typed.

In MetaPRL, transformations are specified with rewrites and judgments are specified with inference rules. In order to specify which rules and rewrites to apply under given circumstances one writes MetaPRL tactics, which are written in ML.

Figure 1 presents a rewrite rule used in the process of fixing the order of evaluation and making all intermediate computations explicitly named. Translating this rule to MetaPRL syntax is straightforward, as demonstrated in the figure. The rule itself rewrites a function application so that the function expression is evaluated before its argument and each of these intermediate computations is stored in a named variable. We assume that expression $e2[v]$ has already been “named”.

Mathematical specification:

- 1: $\text{Name } v = \text{Apply}\{e1; \text{args}\} \text{ in } e2[v] \longleftrightarrow$
- 2: $\text{Name } f = e1 \text{ in}$
- 3: $\quad \text{NameArgs } x = \text{args} \text{ in}$
- 4: $\quad \quad \text{let } v = \text{Apply}\{f; x\} \text{ in } e2[v]$

MetaPRL code:

- 1: $\text{Name}\{\text{Apply}\{e1; \text{args}\}; v. e2[v]\} \langle \dashrightarrow \rangle$
- 2: $\text{Name}\{e1; f.$
- 3: $\quad \text{NameList}\{\text{args}; x.$
- 4: $\quad \quad \text{Let}\{\text{Apply}\{f; x\}; v. e2[v]\}\}$

Figure 1. Function Application Rewrite

Assumptions:

- 1: $\Gamma \vdash \text{TyEqual}\{ty_1; t_1\}$
- 2: $\Gamma, v_1 : ty_1 \vdash$
 $\quad \lambda(v_2 : ty_2, \dots, v_n : ty_n).e : (t_2 \rightarrow \dots \rightarrow t_n \rightarrow t_e)$

Goal:

- 1: $\Gamma \vdash \lambda(v_1 : ty_1, \dots, v_n : ty_n).e : (t_1 \rightarrow \dots \rightarrow t_n \rightarrow t_e)$

Figure 2. Lambda Introduction Inference Rule

Figure 2 illustrates the mathematical specification of a type checking rule. Again, the MetaPRL syntax is straightforward but it has been left out for space reasons. In this rule we specify that for a function to be well-typed its first argument must match the first argument type, and the function that takes one fewer argument must match the type with one fewer argument type. There is also a base case rule, which applies when all arguments have been checked, that verifies that the body of the function has type t_e , but we have not included it in the figure.

Using rewrites, inference rules, and tactics we can transform a program from the source language all the way to assembly code, writing the stages of our compiler in a succinct, mathematical language.

3. Preliminary results

In order to investigate the use of a formal programming environment for writing compilers, we used MetaPRL to write a complete compiler for a small, untyped, ML-like language [2]. Every stage of the compiler, from parsing to code generation, was written using term rewrites. The result is a compiler whose trusted core is dramatically smaller than usual, relying only on several hundred rewrite rules instead of tens of thousands of lines of code. Furthermore, these rewrite rules are very similar to the rules one would write to specify such a compiler mathematically, greatly increas-

ing our confidence in the correctness of the compiler. Correctness could also be affected by MetaPRL’s rewriting engine, but we believe that years of extensive tests and checks for correctness have made it trustworthy.

The success of this experiment motivated us to further explore this methodology by building a compiler for a more complex language. While the experimental compiler was for an untyped language, the new compiler is for a typed language, which means that we must formally specify type checking and inference. We are also attempting to address the problem of extensibility by using a non-traditional compiler design and exploiting the extensible nature of MetaPRL logics. Our compiler is divided into staged pipelines, which means that new features can be added as new pipelines that exploit existing pipelines without changing the semantics of existing features. The core pipeline of the compiler supports only a minimal lambda calculus – all other features are extensions.

It should be noted that our approach to building compilers is not without its own challenges. Some program transformations, such as non-local optimizations and transformations involving capturing substitution of variables, can be difficult to specify using only rewrites. Even so, MetaPRL does allow us to fall back to ML code if it is needed.

4. Future Work

The implementation of the formal compiler is currently under development. Once the core pipeline is finished, we will investigate the modularity of the design. We hope to be able to add new features to the source language (arrays, for example) without changing the core module’s files or interfering with its functionality.

Furthermore, we would like to use this compiler as a platform for experimenting with features like process migration and distributed speculative execution, which would help programmers to build reliable distributed applications more easily.

References

- [1] J. Hickey, A. Nogin, R. L. Constable, B. E. Aydemir, E. Barzilay, Y. Bryukhov, R. Eaton, A. Granicz, A. Kopylov, C. Kreitz, V. N. Krupski, L. Lorigo, S. Schmitt, C. Witty, and X. Yu. *MetaPRL — A modular logical environment*. Accepted to the TPHOLs 2003 Conference, 2003.
- [2] J. Hickey, A. Nogin, A. Granicz, and B. Aydemir. *Formal compiler implementation in a logical framework*. In *MERLIN, Second ACM SIGPLAN Workshop on MEchanized Reasoning about Languages with varTable biNding*, 2003.
- [3] J. D. Smith. *Fault tolerance using whole-process migration and speculative execution*. Master’s thesis, California Institute of Technology, 2003.