Building Reliable Compilers with a Formal Methods Framework

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

1. Introduction

We describe an ongoing research effort into the engineering of reliable compilers by building a compiler within the MetaPRL formal programming environment. 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.

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 in an extended variant of OCaml.

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. Trust

We define the *trusted code base* of a compiler as the code that must necessarily be implemented correctly for the compiler to be trusted. Bugs in code outside of the trusted code base may cause compilation to fail, but it can never cause a successful compilation to produce incorrect output.

Traditionally, the separation between trusted and untrusted code has been recognized only vaguely, if at all. Our technique has the advantage of providing a sharp distinction between them. All code manipulation must be done with rewrites, so all code outside of rewrites is excluded from the trusted code base. Additionally, rewrites comprise only a small fraction of the size of the complete code base, which means the trusted code base can be kept quite small.

4. 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. 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 lines of 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 increasing our confidence in the correctness of the compiler.

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 nontraditional 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.

5. 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.