Applications of Metatheory: Verification of Compiler Optimisations

Richard Warburton
University of Warwick
rlmw@dcs.warwick.ac.uk

Sara Kalvala
University of Warwick
sk@dcs.warwick.ac.uk

A commonly raised critique of the use of formal methods is the practicality of the methodologies and techniques proposed. Frequently there is evidence to support this perspective: claims that are sometimes 'verified' don't hold true, and techniques are devised that prove properties of programs that seldom cause a mainstream programmer too much difficulty. Such criticisms can be addressed in a constructive way, by building tools and proving theorems that affect general use.

Programming language metatheory offers an interesting avenue towards this goal. The approach we concentrate on is verification of compiler optimisations. In this approach, given some optimisation, one ensures that for all programs it is applied to the transformed program will preserve the semantics of the input program. In order to verify language transformations we start with a formal semantics of the programming language that is being used. In some earlier works, the language being optimised has been simplified in order to reduce the effort required to verify or validate the soundness of a given optimisation, reducing the usefulness of the approach. This is one of our motivations for automating programming language semantics and metatheory.

We present our optimisation specifications in TRANS - a declarative language that combines elements of model checking, logic programming and rewriting. A central component of our approach is how to use a formalisation of the language in a theorem prover to build a system that allows one to apply such specifications to real world programs.

In his PhD thesis, Norrish presents a formal semantics for the C programming language, and proves type preservation and safety. These last two results are of interest to us, since they allow a proof of transformation to be able to rely on certain base properties of the language. Nipkow and Klein present a mechanised semantics for a Java-like language, known as Jinja, for which they have developed bytecode semantics. These are in the form of a function that denotes a bytecode operator in the context of the state of the virtual machine by mapping it onto the state of the virtual machine and are used in our work on verification of optimisations on Java Bytecode. In parallel with these theoretical advances the Sable Group at McGill University has developed the Soot framework. This offers a high level intermediate representation, Jimple, against which program transformations can be applied.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]. . . $5.00

The compiler that is presented here, Rosser, comprises three components. A meta-compiler translates TRANS specifications to produce the optimising phase. The program generated relies on a runtime structure, common to all optimisations built on top of the Soot framework. After an optimisation phase is generated by the compiler from a specification it can be loaded into the runtime framework and then applied to a program via the Soot framework.

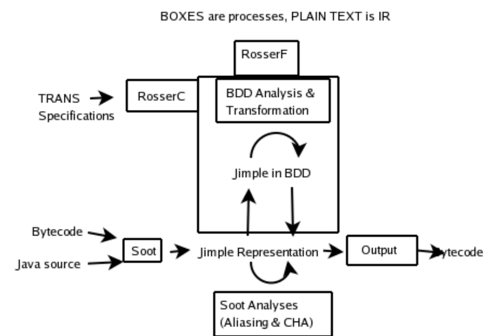

Figure 1. Overall System Architecture

The meta-compiler, RosserC, refines the transformations by

1. refining the logic that specifies the optimisation's analysis to a minimal set of connectives,
2. type checking the language to ensure specifications are well formed, and
3. rewriting the aliasing predicates to allow the use of efficiently computable conservative approximations,

before outputting a compiled optimiser. The runtime framework, RosserF, alters the representation of the program within the Soot framework, so that programs are represented using BDDs, relationally rather than through a traditional object orientated approach.

In terms of formalisation we are developing machine checked proofs of soundness for TRANS specifications. We base our model of what a program is on the semantics for Jinja. Grounded in Jinja, a definition for Control Flow Graphs is specified and a Denotational style semantics for TRANS then builds on that definition. The semantics for TRANS is defined with respect to a Control Flow Graph, a valuation and a TRANS specification. The semantics specifies the changes made to the Control Flow Graph by a valuation, if the TRANS side condition holds true, as well as whether a side-condition holds true for a given valuation.

This work provides evidence that it is possible to start with a formalisation as a basis towards building a practical tool for program development.