Analysis of MiniJava programs via translation to ML

Analysis of MiniJava Programs via Translation to ML

Martin Mariusz Lester
Department of Computer Science
University of Reading
United Kingdom
m.lester@reading.ac.uk

Abstract

MiniJava is a subset of the object-oriented programming language Java. Standard ML is the canonical representative of the ML family of functional programming languages, which includes F# and OCaml. Different program analysis and verification tools and techniques have been developed for both Java-like and ML-like languages. Naturally, the tools developed for a particular language emphasise accurate treatment of language features commonly used in that language. In Java, this means objects with mutable properties and dynamic method dispatch. In ML, this means higher order functions and algebraic datatypes with pattern matching.

We propose to translate programs from one language into the other and use the target language's tools for analysis and verification. By doing so, we hope to identify areas for improvement in the target language's tools and suggest techniques, perhaps as used in the source language's tools, that may guide their improvement. More generally, we hope to develop tools for reasoning about programs that are more resilient to changes in the style of code and representation of data. We begin our programme by outlining a translation from MiniJava to ML that uses only the core features of ML; in particular, it avoids the use of ML's mutable references.

CCS Concepts • **Software and its engineering** → *Formal methods*; *Object oriented languages*; *Functional languages*;

Keywords Java, ML, automated verification, static analysis, program transformation

ACM Reference Format:
Martin Mariusz Lester. 2019. Analysis of MiniJava Programs via Translation to ML. In *Formal Techniques for Java-like Programs (FTfJP'19), July 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3340672.3341119

FTfJP'19, July 15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6864-3/19/07…$15.00
https://doi.org/10.1145/3340672.3341119

1 Motivation

Tools for program analysis and verification have developed rapidly since the success of Microsoft's SLAM driver verification project [3]. A range of complementary and overlapping techniques and technologies have gained prominence, such as abstract interpretation, model-checking, CEGAR and SMT solvers. All provide some way of bounding potentially infinite behaviours in a program or avoiding state space explosion.

Many of the biggest successes have been in the world of traditional imperative programs. Idiomatic C programs make *comparatively* little use of dynamic memory allocation, but may control their behaviour through intricate use of bit-level manipulation and values of complex combinations of flags and other variables. Bounded model-checking using SMT solvers has been particularly successful here [6].

There has also been some success in dealing with object-oriented programs, such as those written in Java [8], and functional programs [10], written in ML or Haskell.

The challenges for handling idiomatic programs written in these paradigms are different. In Java, allocation of objects on the heap is very common. Use of dynamic method dispatch is central to writing idiomatic Java code of any complexity. This means that, even for simple programs, accurate modelling of program control flow requires good modelling of the heap, combined with context sensitivity to match method calls and returns. (In C programs, the equivalent problem of tracking function pointers stored at heap-allocated memory locations still arises, but less frequently.) However, this may not always be important for program verification, as in a well-designed object-oriented program (or at least one that obeys the Liskov Substitution Principle), methods of a subclass that override methods in the superclass will usually satisfy a stronger specification than the method they override. Thus, for many verification problems, it is not necessary to know exactly which subclass method is being called.

In functional languages, the use of higher order functions is similarly prevalent. Conceptually, the difficulty they present is similar to dynamic method dispatch in object-oriented programming, but the complexity of analysis required is often greater. Firstly, accurately tracking flow control for higher order functions requires tracking of more levels of calling context. Secondly, the same functionals are often used in a wide variety of unrelated situations, so type

information cannot reliably be used to delineate and partition their uses. Furthermore, for the same reason, determining the actual results of functionals is more important for accurate program analysis. Consequently, many analyses for functional programming languages emphasise accurate modelling of control flow for higher order functions. In contrast, they often neglect or ignore mutable state, as its use is prohibited in Haskell (other than through monads) and discouraged in ML.

2 Goals

Because Java and ML have different feature sets, it is difficult to apply an analysis designed for one language to a program written in the other. But by doing so, we may gain some insight into our tools and techniques. We may discover that techniques developed in one community would be useful to the other. Or the inability of one community's tool to handle programs from the other may motivate improvements to the tool. In particular, the introduction of lambda expressions to Java 8 may make it more important for Java tools to be able to reason about higher order functions in programs written in a functional style [7].

We propose to begin this exploration by translating Java programs into ML, so that they may be analysed by tools written for functional programs. In order for the translation to be manageable, we focus on translating the MiniJava subset of Java. So that our translated programs may be used with as many tools as possible, we use only the core features of the language, namely recursive functions, algebraic datatypes (including lists) and pattern-matching. In particular, we avoid the use of references (mutable variables). Subject to these constraints, we aim to be as idiomatic as reasonably possible in our translation.

MiniJava is a subset of Java introduced in Appel and Palsberg's book *Modern Compiler Implementation in Java*[2]. Types in MiniJava are limited to `int`, `boolean`, arrays of `int` and object types corresponding to any classes defined in the program. Java features omitted from MiniJava include interfaces, explicit casts, exceptions, visibility modifiers, generics and reflection. The combination of features is expressive enough for writing idiomatic object-oriented programs, but constrained enough to support easy compilation, analysis or transformation.

3 Translation

Statements and expressions. Each Java statement becomes a let-binding, with the "current" program state being used in the bound expression and the "next" program state being the newly bound variables. The style of the resulting code is similar to Administrative Normal Form [4].

Mutable state. The mutable state of a Java program is split into two parts: heap-allocated objects and method-local variables. As the number of local variables in any method is fixed, the local variables can be encoded as a fixed-size tuple of variable values. The heap is a map from pointers to objects. Pointers can be encoded using any datatype that supports the operations required for a name, namely comparison for equality and creation of fresh names. The simplest choice is to use unbounded integers starting at 0, allocating integers sequentially as fresh pointers. Any encoding of maps can be used, but the choice will impact the analysis of the translated program.

Objects and subclasses. Java objects are encodable as a tuple combining their methods (which become ML functions) and their properties (which become either `int`s, `bool`s or `int`s encoding object pointers). Member lookup simply becomes selection of an element from the tuple. Property update requires replacing the whole object in the map encoding the heap. Subclassing could be handled using row-level polymorphism for records [15], as in OCaml's objects. As this is not part of Standard ML, we instead encode an object as a tuple combining its members *and* an Option for any subclass members. The type of the Option is then an algebraic sum over all possible subclasses.

4 Related Work

Program transformation is often used for removal of more complex features of a language [5], or translation to a simpler language, so that the verification tools need only handle a smaller number of language features. Notably, the Jimple [14] intermediate language for Java used by Soot is deliberately simpler than Java bytecode. Such transformations are often avoided, as they hide the structure of a program, confounding analysis. Indeed, attempting to recover this structure is a key step in analysis of compiled programs [9].

Previous work considers analysis of functional programs written in Haskell via translation to C using the compiler JHC and application of the symbolic execution tool Klee [1]. We are not aware of any work in the reverse direction, presumably because of the relative immaturity of tools for functional languages. Tools for analysing ML programs are based around a variety of different techniques, such as model-checking of Higher Order Recursion Schemes (MoCHi [13]), refinement type inference (DSolve [12]) and algorithmic game semantics (SyTeCi [11]), but there is no clear leader.

5 Status and Future Work

We are currently implementing the translation. The starting point for our work is a toy MiniJava compiler used to teach a module on compilers at the University of Reading. The next step will be to compare Java analysis tools on MiniJava programs with ML program tools on the translated programs.

We expect that they will be reasonably accurate until they have to reason about values retrieved from the heap, however we choose to encode it.

References

[1] Mario Alvarez-Picallo. 2015. MPRI Internship Report: Verification by compilation of higher-order functional programs. (2015).

[2] Andrew W. Appel and Jens Palsberg. 2002. *Modern Compiler Implementation in Java, 2nd edition.* Cambridge University Press.

[3] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. 2004. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings (Lecture Notes in Computer Science)*, Eerke A. Boiten, John Derrick, and Graeme Smith (Eds.), Vol. 2999. Springer, 1–20. https://doi.org/10.1007/978-3-540-24756-2_1

[4] Robert Cartwright (Ed.). 1993. *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993.* ACM. http://dl.acm.org/citation.cfm?id=155090

[5] Wontae Choi, Baris Aktemur, Kwangkeun Yi, and Makoto Tatsuta. 2011. Static analysis of multi-staged programs via unstaging translation. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 81–92. https://doi.org/10.1145/1926385.1926397

[6] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science)*, Kurt Jensen and Andreas Podelski (Eds.), Vol. 2988. Springer, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

[7] David R. Cok. 2018. Reasoning about functional programming in Java and C++. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, Amsterdam, Netherlands, July 16-21, 2018*, Julian Dolby, William G. J. Halfond, and Ashish Mishra (Eds.). ACM, 37–39. https://doi.org/10.1145/3236454.3236483

[8] Lucas C. Cordeiro, Daniel Kroening, and Peter Schrammel. 2018. Benchmarking of Java Verification Tools at the Software Verification Competition (SV-COMP). *ACM SIGSOFT Software Engineering Notes* 43, 4 (2018), 56. https://doi.org/10.1145/3282517.3282529

[9] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction, Austin, TX, USA, February 5-6, 2017*, Peng Wu and Sebastian Hack (Eds.). ACM, 131–141. https://doi.org/10.1145/3033019

[10] Marco Gaboardi, Suresh Jagannathan, Ranjit Jhala, and Stephanie Weirich. 2016. Language Based Verification Tools for Functional Programs (Dagstuhl Seminar 16131). *Dagstuhl Reports* 6, 3 (2016), 59–77. https://doi.org/10.4230/DagRep.6.3.59

[11] Guilhem Jaber. 2018. SyTeCi: Towards automation of contextual equivalence for higher-order programs with references. (2018).

[12] Ming Kawaguchi, Patrick Maxim Rondon, and Ranjit Jhala. 2010. Dsolve: Safety Verification via Liquid Types. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.), Vol. 6174. Springer, 123–126. https://doi.org/10.1007/978-3-642-14295-6_12

[13] Ryosuke Sato, Hiroshi Unno, and Naoki Kobayashi. 2013. Towards a scalable software model checker for higher-order programs. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013, Rome, Italy, January 21-22, 2013*, Elvira Albert and Shin-Cheng Mu (Eds.). ACM, 53–62. https://doi.org/10.1145/2426890.2426900

[14] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, Stephen A. MacKay and J. Howard Johnson (Eds.). IBM, 13. https://dl.acm.org/citation.cfm?id=782008

[15] Mitchell Wand. 1989. Type Inference for Record Concatenation and Multiple Inheritance. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989.* IEEE Computer Society, 92–97. https://doi.org/10.1109/LICS.1989.39162