

Tinyrossa: A Compiler Framework for Vertical, Verified Construction of Smalltalk VMs

Jan Vraný

LabWare

Dundee, United Kingdom

jan.vrany@labware.com

Boris Shingarov

LabWare

Ottawa, ON, Canada

shingarov@labware.com

ABSTRACT

This talk presents Tinyrossa, an experimental JIT- and AoT-compilation framework which is part of our larger “Smalltalk-25” program of research towards constructing dynamic-language VMs using formal methods.

As far as compilation per se, Tinyrossa does not make novel contributions; it is a traditional compiler that performs IL-to-IL and IL-to-Native code transformations. The main point of our work is to treat existing compilation algorithms within a mathematically-rigorous framework of executable logical specifications.

To that end, each code transformation is endowed with a “Justification” (a certificate proving the transformation’s soundness). Justifications are computed by *MachineArithmetic*, our SMT-based theorem prover written in Smalltalk.

Tinyrossa is an ongoing research effort. In this workshop talk, we present our current status using an example walking through the compilation of a simple method down to optimized native RISC-V code.

CCS CONCEPTS

• **Software and its engineering** → **Semantics; Runtime environments**; • **Theory of computation** → **Operational semantics**.

KEYWORDS

Smalltalk, Virtual Machine, Formal Verification, Program Synthesis

ACM Reference Format:

Jan Vraný and Boris Shingarov. 2024. Tinyrossa: A Compiler Framework for Vertical, Verified Construction of Smalltalk VMs. In *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming (Companion ’24)*, March 11–15, 2024, Lund, Sweden. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3660829.3660838>

1 BACKGROUND

Rigorous engineering of compilers using formal methods has seen tremendous progress and has become an established discipline.

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. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Companion ’24, March 11–15, 2024, Lund, Sweden

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0634-9/24/03

<https://doi.org/10.1145/3660829.3660838>

Unfortunately, the art of implementation of dynamic programming languages has not benefited from this progress. Attempts at formalization of Smalltalk or Java VMs have not progressed beyond idealized models [5, 6, 10, 13]. In the authors’ practical experience working on the industry’s major Smalltalk and Java VMs, the methods of construction of said VMs remain completely informal. Worse yet: to construct a verified compiler, one needs a tall stack of tools and technologies, none of which are available in any existing Dynabook-style environment.

Frustrated with this situation, we have embarked upon *Smalltalk-25*, a scientific *Programm* of research towards the synthesis of Smalltalk VMs by formal methods. In this talk we present *Tinyrossa*, the *compiler* component of Smalltalk-25. Before we can concentrate on *Tinyrossa* proper, we need to say a few words about *MachineArithmetic*, the proof-theoretic foundation of Smalltalk-25. There are roughly two schools of automated proof: type-theory-based (e.g. Coq) and satisfiability-based (e.g. Z3). Accordingly, compiler verification can follow the former (e.g. CompCert) or the latter (e.g. CN). Vazou et al. [19] have recently shown that Natural Deduction can be encoded as SMT-checkable programs, thus enjoying the benefits of both the TT and SMT approach. *MachineArithmetic* [15] implements Vazou et al’s algorithm in Smalltalk; this design decision largely dictates design decisions in the rest of Smalltalk-25.

2 TINYROSSA

To avoid infinite inflation of scope of the project, we decided to limit ourselves to one pre-existing, well-understood compiler design which we are familiar with — namely that of the OMR Testarossa compiler — and concentrate on the formal aspect, cutting out issues such as IR design, practicality, scalability in production *etc* as orthogonal and pertaining to Testarossa proper. In particular, we reuse the Testarossa Intermediate Language (TRIL), and to some degree follow the overall architecture of the compiler.

Tinyrossa is a traditional direct-style compiler. It traverses the IR tree looking for matching rules. Rules specify “transformations”, which fall into two categories:

- “optimizations” (TRIL to TRIL), and
- “evaluators” (TRIL to non-TRIL).

Every transformation is endowed with a “Justification”. Let’s look at one example of TRIL-to-TRIL optimization to see how these justifications are produced.

2.1 Example: Sum of Arithmetic Series

Consider the following Smalltalk program for calculating the sum of arithmetic sequence $1 + 2 + \dots + n$ by naïve iteration:

```
sum: n  
| x r |
```

```

x := n.
r := 0.
[ x > 0 ] whileTrue:[
  r := r + x.
  x := x - 1.
].
^ r

```

which takes $O(n)$ additions and conditional jumps. We want to justify replacing this loop by the purely-arithmetic $O(1)$ expression $n * (n + 1) / 2$.

For the sake of illustration, let's imagine that the front-end (the details of which are not relevant to our discussion) has produced the following TRIL program:

```

N000 bbstart <sum>
N019 ificmpne BB_002
N017      iload n
N018      iconst 0
N005 ireturn
N004      iconst 0
N001 bbend </sum>
N006 bbstart <BB_002>
N014 ireturn
N013      iadd
N008      iload n
N012      icall sum
N011      isub
N008      ==> iload n
N010      iconst 1
N007 bbend </BB_002>

```

Tinyrossa transforms this program into MachineArithmetic's "semantic core" IL using an explicit [Plotkin-style small-step] operational semantics of TRIL. The latter consists of two parts: (1) the structural part (how to traverse the tree), and (2) a library of opcodes expressed as semantic-core functions:

```

reflect iconst : bv32 => bv32
let iconst = (const) => {
  const
};

reflect iadd : bv32 => bv32 => bv32
let iadd = (e1, e2) => {
  bvadd32(e1, e2)
};

reflect ificmpne : bv32 => bv32 => bool
let ificmpne = (e1, e2) => {
  bvne32(e1, e2)
};

```

In real operation of Tinyrossa, semantic-core programs are Smalltalk objects representing ASTs of the prover IL, but for experiments and illustrations we notate it in the concrete syntax of Jhala–Vazou's didactic paper [11]. The traversal transforms the two basic blocks into two functions:

```
val BB_002 : x:bv32 => bv32
```

```

let BB_002 = (x) => {
  let N008 = iload(x);
  let N010 = iconst(bv32(1));
  let N011 = isub(N008, N010);
  let N012 = sum(N011);
  let N013 = iadd(N008, N012);
  ireturn(N013)
};

reflect sum : x:bv32 => bv32
let sum = (x) => {
  let N017 = iload(x);
  let N018 = iconst(bv32(0));
  let cond = ificmpne(N017, N018);
  if (cond) {
    BB_002(x)
  } else {
    let N004 = iconst(bv32(0));
    ireturn(N004)
  }
};

```

By some trivial inlining and simplification, these two functions, combined with the above opcode-semantics functions (iconst etc), get reduced to:

```

reflect sum : n:bv32 => bv32 / [n]
let rec sum = (n) => {
  let cond = bvne32(n, bv32(0));
  if (cond) {
    let n1 = bvsub8(n, bv8(1));
    let t1 = sum(n1);
    bvadd8(n, t1)
  } else {
    bv8(0)
  }
};

```

To prove that this recursive function terminates and returns $n * (n + 1) / 2$, which is expressed by the following theorem:

```
val thm_sum : n:bv32 => void[_] 2*(sum value: n) ==
                                   (n+1 * n) / [n]
```

we provide an explicit proof term:

```

val thm_sum : n:bv32 => void[_] 2*(sum value: n) ==
                                   (n+1 * n) / [n]

let rec thm_sum = (n) => {
  let base = n == bv32(0);
  if (base) {
    '()
  } else {
    let n1 = bvsub(n, bv32(1));
    let thm = thm_sum(n1);
    '()
  }
};

```

How does MachineArithmetic typecheck this program? Because the explicit type of `sum()` does not tell us anything, we are forced to symbolically execute `sum(n)`. We do this by unfolding `sum's`

body into an execution trace. Without knowing n , this trace is an infinite tree because of unbounded recursion (Figure 1). Turchin’s “Supercompilation” idea [7, 16, 17] is to *unify* the subtree at $f(n-1)$ with the whole tree at $f(n)$ modulo substituting $n-1$ for n . Our goal is not only to ensure the arithmetic equality $2 * \text{sum}(n) = n * (n+1)$, but also to guarantee termination¹ of the program. We proceed by induction on n , case-splitting at

```
if (cond) {...} else {...}
```

into the induction base and the induction step. This *splitting* into Dijkstra guarded form [8] is crucial because it is what guarantees that the symbolic execution itself terminates [19].

The base case terminates immediately, yielding a trivial Z3 tautology. In the inductive case, we assume the $(n-1)$ induction hypothesis

$$2 * \text{sum}(n1) = n1 * (n1 + 1) \text{ where } n1 = n - 1$$

and try to prove that it entails the conclusion $2 * \text{sum}(n) = n * (n+1)$ for $n \neq 0$. *MachineArithmetic* generates two Z3 constraints. The first constraint guarantees termination using well-foundedness in bit-vectors under “unsigned less-than” (because any “unsigned-decreasing” sequence — even if it may wrap under the signed interpretation — will reach zero):

```
(= (= n #x00) false)
(= n1 (bvsb n #x01))
==>
(bvult n1 n)
```

The second “Unsat Candidate” represents the actual arithmetic equality:

```
(= (= n #x00) false)
(= n1 (bvsb n #x01))
(= (sum n1)
    (bvmul (bvadd n1 #x01) n1))
==>
(= (bvmul #x02 (sum n))
    (bvmul (bvadd n #x01) n))
```

Both of these constraints are easily discharged by Z3.

2.2 ISA and Relaxed Memory Semantics

Justifying TRIL-to-Native transformations is much more involved. Existing work on mechanized reasoning about machine architecture, (such as CompCert[12], VST[2], CakeML[9], and many others), has traditionally been using ad-hoc hand-written axiomatizations to serve as the definition of [modest fragments of] the architecture. Surprisingly, real processor semantics are much more intricate than these idealized models. Recently, Armstrong et al. [3] and Armstrong et al. [4] made progress toward replacing these models by *complete* and *authoritative* specifications. These specifications integrate sequential aspects of the ISA with non-deterministic factors such as relaxed memory, instruction fetch etc.

Tinyrossa justifies TRIL-to-Native transformations via abstract interpretation of Armstrong et al.’s Jib IL. In Tinyrossa, Armstrong’s “footprint” is a program in the *MachineArithmetic* core-language. This program is checked to be equivalent to the TRIL program, by

¹Of course we are interested in *total* correctness because otherwise we could prove *anything* by a circular argument

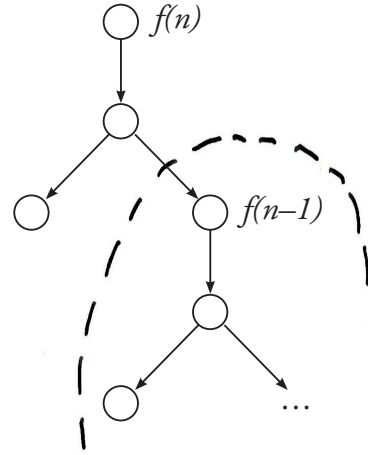


Figure 1: Example of supercompilation: finding self-similarity in a symbolic execution trace.

the same process of reflection into SMT that we saw in the previous section.

2.3 Demo

In the talk, we give a live walk-through another example of TRIL-to-TRIL transformation: a strength reduction from MUL by a power-of-2 to a SHIFT.

Next, we demonstrate how Tinyrossa spots an actual bug: we look at the “absolute value” function:

$$\text{abs}(x) = \begin{cases} x, & x \geq 0 \\ 0 - x, & x < 0 \end{cases} \quad (1)$$

and try to prove the invariant that $\forall x. \text{abs}(x) \geq 0$.

We may be tempted to write the following TRIL implementation of abs:

```
N000  bbstart <entry>
N016  ifcmpge BB_002
N014    iload x
N015    iconst 0
N007  ireturn
N006    isub
N004      iconst 0
N014      ==> iload x
N001  bbend </entry>
N008  bbstart <BB_002>
N011  ireturn
N010    iload x
N009  bbend </BB_002>
```

but Tinyrossa immediately finds the following counterexample (in the 32-bit integers): $x=16r80000000$. In two’s complement, there are one less positive integers than negative ones, so there is in fact no “correct” `abs()` in machine integers, because `MININT` is a valid argument and `abs(MININT)` is too large to fit in a machine word!

Finally, we give a simple example of TRIL-to-Native emission of RISC-V code.

2.4 Current Status

Tinyrossa is an ongoing research effort. The complete Smalltalk source code for Tinyrossa, MachineArithmetic and all other parts of Smalltalk-25 is available on GitHub under the MIT license.

As a *compiler*, at the time of writing *Tinyrossa* implements a subset of the OMR Testarossa set of opcodes and optimizations. This subset is enough to compile a number of simple test programs for RISC-V and POWER and run them on real hardware, and in simulation (QEMU and gem5) for regression testing in CI. In principle, *Tinyrossa* is portable to any architecture for which there is a formal specification compileable to the Jib IL; this includes ARM and MIPS.

The validation suite for the Justification component follows the general layout of LiquidHaskell’s benchmarks [18]: there is a set of positive tests and a set of negative tests. In each positive test, there is a correctly-specified transformation, and to pass the test the system has to prove correctness by showing SMT validity. In each negative test, there is an incorrectly-specified transformation, and to pass the test the system has to refute correctness by finding a counterexample.

At the time of writing, these tests exist and pass for a limited subset of TRIL, using hand-written semantic functions.

Two major remaining/incomplete blocks, are integration with Sewell et al. [14]’s “Ott” semantic tool, and using Alglave et al. [1]’s “Cat” memory semantics. Once these pieces are complete, it will allow us to replace the hand-written fragment of TRIL semantics by a complete Ott TRIL specification, and use Tinyrossa for discovery of race-condition bugs.

REFERENCES

- [1] Jade Alglave, Patrick Cousot, and Luc Maranget. 2016. Syntax and semantics of the weak consistency model specification language cat.
- [2] Andrew W. Appel. 2014. *Program logics for certified compilers*.
- [3] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*. Proc. ACM Program. Lang. 3, POPL, Article 71.
- [4] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. 2023. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models (extended version). *Formal Methods in System Design* (2023).
- [5] Robert Atkey. 2008. *CoqJVM: An Executable Specification of the Java Virtual Machine Using Dependent Types*. Springer Berlin Heidelberg, Berlin, Heidelberg, 18–32.
- [6] Gilles Barthe, Guillaume Dufay, Line Jakubiec, Bernard P. Serpette, and Simão Melo de Sousa. 2001. A Formal Executable Semantics of the JavaCard Platform. In *Proceedings of the 10th European Symposium on Programming Languages and Systems (ESOP '01)*. Springer-Verlag, London, UK, 302–319. <http://dl.acm.org/citation.cfm?id=645395.757559>
- [7] Maximilian Bolingbroke and Simon Peyton Jones. 2010. Supercompilation by Evaluation.
- [8] Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of ACM* (1975).
- [9] Anthony Fox, Magnus O. Myreen, Yong Kiam Tan, and Ramana Kumar. 2017. Verified compilation of CakeML to multiple machine-code targets. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017)*.
- [10] David A. Greve. 1998. Symbolic Simulation of the JEM1 Microprocessor. In *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design (FMCAD '98)*.
- [11] Ranjit Jhala and Niki Vazou. 2021. Refinement Types: A Tutorial. *Foundations and Trends in Programming Languages* 6, 3–4 (2021), 159–317.
- [12] Xavier Leroy. July 2008. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2008), 107–115.
- [13] H. Liu and J. S. Moore. 2003. Executable JVM Model for Analytical Reasoning: A Study. In *Workshop on Interpreters, Virtual Machines and Emulators*. ACM SIGPLAN, San Diego, California, USA.
- [14] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. 2010. Ott: Effective Tool Support for the Working Semanticist. *Journal of Functional Programming* (2010).
- [15] Boris Shingarov. 2022. Live Proof by Induction. In *FAST Workshop on Smalltalk Related Technologies*.
- [16] Morten Heine Sørensen. 1996. Turchin’s Supercompiler Revisited.
- [17] V.F. Turchin. 1986. The concept of a supercompiler. In *TOPLAS*.
- [18] Niki Vazou. 2016. *Liquid Haskell: Haskell as a Theorem Prover*. Ph. D. Dissertation. University of California, San Diego.
- [19] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *Proc. ACM Program. Lang.* 2, POPL, Article 53 (dec 2017), 31 pages.

Received 2024-01-22; accepted 2024-02-05