



Towards a Dynabook for verified VM construction

Boris Shingarov^a, Jan Vraný^b^a LabWare, 21 Aintree Pl., Ottawa, K2M 2G5, ON, Canada^b LabWare, 15 Seafield Road, Dundee, DD1 4NR, United Kingdom

ARTICLE INFO

Keywords:

Virtual machine
Formal verification
Program synthesis

ABSTRACT

We present *Smalltalk-25*, a scientific *Program* of research towards the synthesis of Smalltalk VMs by formal methods. We approach this through a Dynabook-style implementation of Hilbert's "manipulating proofs as mathematical objects". We begin by implementing an automated-reasoning engine based on refinement types. Having that foundation laid, we build a program logics for a simple "model" language; a framework for reasoning about a real intermediate language by translating into the model language; an ahead-of-time bootstrapper; and a framework for debugging thus-constructed VMs.

Our *Program* is an ongoing research effort; this paper provides a report of its current status. Every part of our system is implemented in Smalltalk and available from GitHub under the MIT license.

1. Background

The work presented in this paper was motivated by the authors' decades of practical engineering of the industry's major Java and Smalltalk virtual machines. Our experience suggests that the methods of construction of these VMs are woefully inadequate. It makes little difference whether we program using the straightforward notation of C/C++/Assembly (such as in OpenJ9 or HPS) or a Smalltalk-like syntax (such as in Squeak/OpenSmalltalk VM). We believe the root of the problem has to do with ignoring the progress in programming language semantics during the last fifty-some years. In particular, the customary reasoning over an implementation's correctness is completely manual: when an engineer writes a piece of VM code, the argument as to why that code works exists solely within the engineer's mind (or at best, as an informal comment). This is in contrast to programming as writing programs that manipulate arguments according to Hilbert's idea of proof being just another mathematical object.

Consider for example the following assignment to a local variable *i*:

```
i := 42.
```

The compiler will transform this into some (say, POWER) CPU instructions, perhaps:

```
li    r31, 42
stw   r31, 32(r1)
```

This is little more than macro expansion, in the sense that, nowhere in the compiler's source there is any argument as to why those instructions indeed perform what that *i := 42* is supposed to do.

Opposite to this view of "translation" is "program calculation" or "synthesis", where the *li r31, 42; stw r31, 32(r1)* is actually computed from the *i := 42*, using some formal semantics of the language and of the ISA.

Today, 74 years after Turing's ground-laying paper "Checking a large routine" [1], formally-constructed compilers are no longer a research toy. Just like we have a systematic theory of syntax owing to Chomsky, Backus and many others, so that we have come a long way from ad-hoc parsing to rigorously-engineered grammars, — we have a systematic theory of semantics owing to Scott, Strachey, Floyd, Hoare, Plotkin, McCarthy and many others, so that real, production programming language implementations can now be computed from formal specifications.

In our engineering of real-life industrial Smalltalk and Java VMs, we found that constructing a VM using informal ad-hoc methods, in a modern computational environment is not just error-prone or tedious, but virtually impossible. "Classical" architectures (e.g. from IBM S/360 onward) assume consistent, deterministic behavior among executions and among implementations, and the linear array of core-memory cells as the only conceivable "structure". From this century's perspective, this is only one point in the design space. For example, most of today's architectures are only weakly-consistent. Now suddenly *testing* is no longer a reliable indicator whether the program behaves as expected: a test may pass on some particular implementation of the ISA hitting

* Corresponding author.

E-mail addresses: shingarov@labware.com (B. Shingarov), jan.vrany@labware.com (J. Vraný).

some particular timing, but this tells us nothing about another vendor's implementation or a different timing in a race. So *tests* must be replaced with *proofs* guaranteeing that such-and-such holds *for all conforming implementations* and *for all timings*.

Therefore, we set out to build a development environment for systematic, formal construction of VMs using standard methods well-established in the field of verified compilers. We do not pretend that any of our mathematics or algorithms are original. Why is it interesting, then, to attempt such a system in Smalltalk, if the field has been so thoroughly developed already?

We see two reasons.

To the authors' best knowledge, Hilbert's "computing of proofs" has never been done in a Dynabook-like medium. Agda, Coq, NuPRL, or any number of others — are all rooted in the LCF/ML tradition, characterized by a negative attitude towards the very concept of "debugging". Diametrically opposite to this, is the Dynabook's goal to afford the mathematician a sensation of immediate, almost tactile, contact with the mathematical construct. So when a proof fails, I want a Smalltalk debugger to open and give me direct access to what went wrong. In this sense manipulating a proof in Smalltalk is a different experience than manipulating a proof in Agda.

The second reason is that, while the "rigorous engineering" discipline has become well-established in building compilers for static languages, applying similar approaches to dynamic languages such as Smalltalk or Java has not progressed beyond idealized models. Production VMs are still completely informal.

Therefore we set out to design a system for formally-verified VM development, which would allow us to achieve the following four goals.

1. As the words "formally verified" suggest, the most immediately-obvious yield from such a system is the assurance guarantees. Less trivially, we know a formal-verification tool works if it deepens our insight into the VM — for example by discovering unexpected edge cases. Finally, if the formal-verification tool is any good, it changes the perspective from which we understand the VM.
2. We want to experience automated reasoning in a Dynabook-like medium. One way we evaluate this, is we take benchmarks used by existing theorem-provers, solve them in Smalltalk-25, and compare the experience.
3. Smalltalk-25 shall enable us to "afford to break our existing toys", i.e. to be able to redesign any aspect of the system without too much pain, so no aspect is set in stone.
4. In Smalltalk-25 we shall be able to program a wider range of computers than the deterministic, sequential, memory-only "universal computers" on which Smalltalk-80 was implemented.

In this paper we give a high-level overview of the Smalltalk-25 *Program* as a whole. In several previous papers [2–18], we concentrated on details of Smalltalk-25's individual constituents (*MachineArithmetic*, *Universally-Live Debugger* etc.), but none of those papers attempt to explain the overall motivation why these components are needed in the first place, or how they fit together. The present paper aims to address this void.

The paper is organized as follows.

To explain the main ideas, we start from a simple example: a program that calculates 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 explain the steps the optimizer uses to justify replacing this loop by the purely-arithmetic $O(1)$ expression $n * (n + 1) / 2$. Each of these "system structure" sections — one per component — follow the same plan: we explain the purpose of the component in the system; give the gist of how it achieves its function; and discuss related work, the current status of our work-in-progress, and evaluation. These sections form the central Part 2 of the paper. In Part 3, we talk about the lessons we learned from our Smalltalk-25 journey up to date. The final Part 4 discusses future work.

2. Structure of the system

Our system comprises the following major components:

- *MachineArithmetic*, a calculus of refinement types;
- *Tinyrossa*, a compiler based on a formal semantics for OMR's Testarossa Intermediate Language (TRIL);
- *PIG*, a superoptimization framework for manipulating formal semantics in the *MachineArithmetic* proof-theoretic calculus;
- The *Universally-Live Debugger*, allowing Smalltalk-style live debugging of the lowest levels of Smalltalk's object engine;
- *Ahead-of-Time*, a native bootstrapper for modular Smalltalk systems.

Fig. 1 shows how these components can work together in one possible Smalltalk-25 system. For simplicity, let us concentrate on the back-end, deferring the discussion of the front-end to Section 2.5. In other words, let us for the moment assume the front-end (1) has produced an intermediate representation in something like the "Testarossa Intermediate Language" (TRIL) of OpenJ9/OMR [19].¹ This initial IR (2) is input into our *Tinyrossa* compiler, which performs (possibly many) TRIL-to-TRIL optimizations (3) followed by TRIL-to-Native CPU code emission transformation (7), just like any traditional compiler does. Unlike in traditional compilers, each *Tinyrossa* transformation is backed by a Justification, which is a lemma in the *MachineArithmetic* theorem prover described in Section 2.1. Justifications of TRIL-to-TRIL transformations (4) are computed from TRIL operational semantics (5) using abstract interpretation as described in Section 2.2. Justification of TRIL-to-Native transformation (7) is done similarly (in principle) but involves target ISA and memory semantics (9), as explained in Section 2.2.4. The result of the TRIL-to-Native transformation — and thus the output of *Tinyrossa* — is native machine code (10). We debug the resulting native machine code running on a particular target (11) by *Universally-Live Debugger* (12).

Let us look at each of these components, in a bottom-up order starting from the foundational level first. The reader shall not be confused by this low- to high-level order of exposition being seemingly "backwards"; the advantage of this order is that at each step we build only on top of already-explained components.

2.1. MachineArithmetic

MachineArithmetic is a Smalltalk implementation of Freeman and Pfenning's Refinement Types [20]. It provides the type-theoretic foundation for the rest of Smalltalk-25. Because we have previously described *MachineArithmetic* in Shingarov [15], here we shall only give a quick recapitulation of the main ideas.

A refinement type conjoins a "coarse type" — such as Integer or String — with a logical predicate which constrains the possible values to a subset. For example, the type $\{x \in \mathbb{Z} \mid x > 0\}$ refines the coarse type $\mathbb{Z}$ to "the set of all positive integers". In Smalltalk we write

¹ For readers familiar with OpenJ9, another way of saying the same thing is that we are for now interested in formally constructing only the counterpart to OMR, not to the "language-specific rest of OpenJ9".

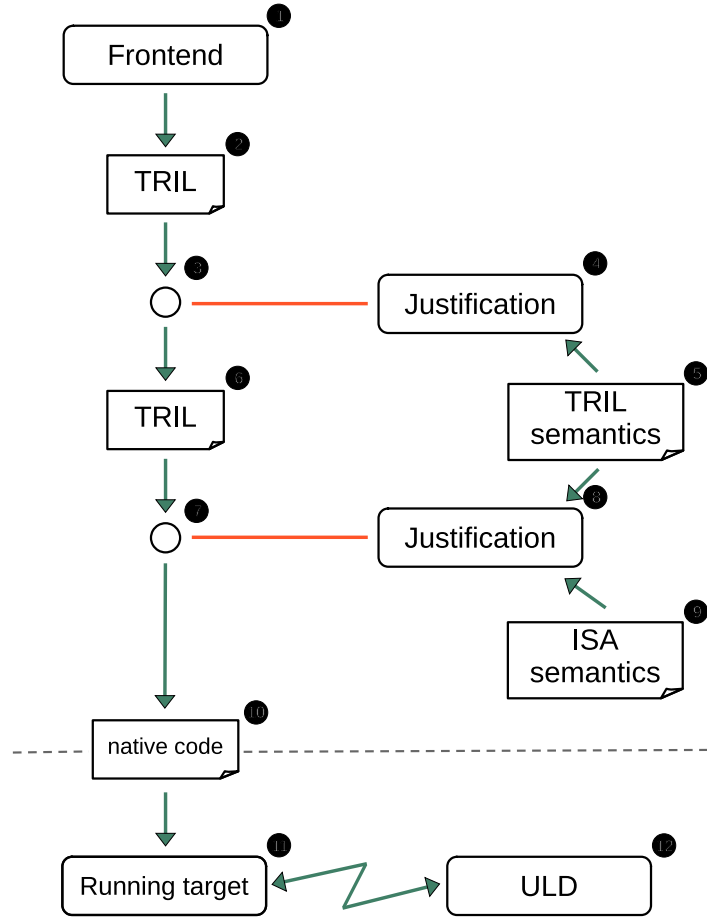


Fig. 1. Structure of one possible Smalltalk-25 system. (Dog-eared boxes are data; rounded boxes are system components; arrows are flow of data; little circles are *Tinyrossa* transformations.).

Integer | [:x | x > 0].

Here #| (pronounced “*such that*”) is a regular Smalltalk-80 binary selector, and its argument is a regular Smalltalk-80 block. *Machine-Arithmetic* performs automated reasoning over higher-order functional constructs involving such blocks by translating them into first-order Horn constraints and then solving the constraints using the Z3 SMT solver [21]. Crucially, these transformations guarantee that we stay within decidable theories so that no call to Z3 ever diverges.

The reasoning over the Smalltalk blocks does not involve any parsing of the Smalltalk source, nor looking at Smalltalk AST — only execution (by the native Smalltalk host) over symbolic arguments. In the above example, the coarse-type declaration $x \in \mathbb{Z}$ — coming from the #|’s receiver Integer — produces the *symbolic constant* x (a live Smalltalk object wrapping around the corresponding Z3 symbolic constant). The block is evaluated on this symbolic constant, yielding a Refinement object wrapping around the Z3 AST “ $x > 0$ ”.

On top of these live objects, we construct a theorem prover in which arbitrary Smalltalk expressions serve as proof terms. The two crucial elements of this prover are (1) a fixpoint Horn solver, and (2) a logical reflection engine which amounts to a Turchin supercompiler. The algorithms in the prover are not original; we follow existing work in [22–24], *mutatis mutandis* due to our use of live Smalltalk expressions amenable to direct debug-time manipulation which gives our system its characteristic Dynabook-like flavor.

In addition to these “proof kernel” elements, our prover also implements two extensions: (1) Vazou–Bakst–Jhala’s “bounded refinement” [25] syntactic sugar, which allows elegant reasoning about mutable state following Filliâtre [26] and Launchbury and Peyton Jones

[27] while staying within the decidable logic of McCarthy arrays [28] as implemented in Z3; and (2) for didactic and for validation purposes, parsers for concrete syntax of benchmarks/examples widely used in the LiquidHaskell community.

2.1.1. Example: Gauss

Let us use our “Gauss” example to illustrate how this works.

The *Tinyrossa* compiler reduces the TRIL representation of the “naïve sum-of-arithmetic-series” program to an ML-like “semantic core” program. (This process is outlined in Section 2.2). In real operation this is an AST in the prover’s abstract “model language” — a Smalltalk object — but for experiments and illustrations we notate it in the concrete syntax of Jhala–Vazou’s didactic paper [29].

```
reflect sum : n:bv32 => bv32 / [n]
let rec sum = (n) => {
  let cond = bvne32(n, bv32(0));
  if (cond) {
    let n1 = bvsub32(n, bv32(1));
    let t1 = sum(n1);
    bvadd32(n, t1)
  } else {
    bv32(0)
  }
};
```

The goal is to justify replacing this by $n * (n + 1) / 2$.

The justification is that the following program (combined with `sum`) typechecks²:

```
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 = bvsub32(n, bv32(1));
    let thm = thm_sum(n1);
    '()
  }
};
```

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 (Fig. 2). Turchin's "Supercompilation" idea [30–32] 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 termination³ 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 [33] is crucial because it is what guarantees that the symbolic execution itself terminates [23].

The base case terminates immediately, yielding a trivial Z3 tautology. In the inductive case, we assume the $(n-1)$ induction hypothesis $2 * \text{sum}(n1) = n1 * (n1+1)$ 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 #x00000000) false)
(= n1 (bvsub n #x00000001))
==>
(bvult n1 n)
```

The second "Unsat Candidate" represents the actual arithmetic equality:

```
(= (n #x00000000) false)
(= n1 (bvsub n #x00000001))
(= (sum n1)
   (bvmul (bvadd n1 #x00000001) n1))
==>
(= (bvmul #x02 (sum n))
   (bvmul (bvadd n #x00000001) n))
```

Both of these constraints are easily discharged by Z3.

² To not distract the reader by irrelevant detail, we write *void* and *'()* where "discarded *int*" / "ignored 0" happen in [29].

³ Of course we are interested in *total* correctness because otherwise we could prove *anything* by a circular argument.

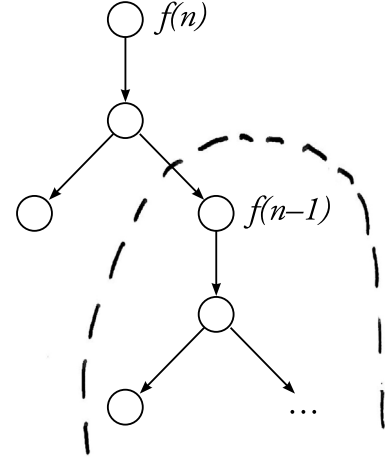


Fig. 2. Example of supercompilation: finding self-similarity in a symbolic execution trace.

2.1.2. An aside

It is worth noting that the above proof is not the same as the `thm_sum` example in [29]. The latter deals with mathematical (infinite-precision) integers. The TRIL program operates on machine integers, i.e. bit-vectors of known length. With respect to proving the arithmetic equality, this does not lead to complications from signed/unsigned two's complement treatment and overflow/wrapping. Termination is only slightly more complicated: in the case of $\mathbb{N}$, any strictly-decreasing sequence reaches zero, whereas in the case of bit-vectors, the same is true of bit-vectors ordered by *unsigned* "<", so we critically depend on the $n \geq 1$ condition in the inductive case. We shall see in Section 2.2.3 how interesting edge cases arise here, and how we investigate them in the Smalltalk debugger.

2.1.3. Related work

The classical tradition in deductive software verification is founded upon type-theoretic theorem provers of the LCF family [34,35] such as Coq or Agda. Characteristic representatives of this tradition would be CompCert [36] or Appel's capital text [37]. On the other hand, there is a large category of verification systems founded upon the recent advances in efficient SMT solving. Examples of this category are diverse, ranging from Boogie [38], Dafny [39], F* [40], to separation logic tools such as Grashopper [41] to large-scale verifiers such as CN [42].

In our earlier paper [43] we described "Verified Linear Smalltalk", exploring the classical type-theoretic approach and building upon the wealth of pre-existing engineering in the external Coq prover. Smalltalk-25 explores the opposite quadrant of this decision space: *MachineArithmetic* is an SMT-based prover implemented in Smalltalk. *MachineArithmetic* does not contribute novel foundational algorithms: our Horn solver follows Cosman and Jhala [22], our Turchin super-compiler is based on Vazou et al. [23], and our bounded-refinement extension follows Vazou et al. [24] and Vazou et al. [25]. The difficulties in debugging proof failure, stemming from the nature of SMT, are discussed in Leino and Pit-Claudel [44], Vazou et al. [23], Tondwalkar [45] and Hallahan et al. [46].

2.1.4. Current status and evaluation

When we initially set out to implement *MachineArithmetic*, we first implemented a parser for the test suite used by the LiquidHaskell prover community. During the early stages of the project, we used these tests as our reference measure of "works". As the codebase stabilized,

our interest moved forward to *MachineArithmetic*'s usefulness as the substrate for higher layers such as *Tinyrossa* and *PIG* (although the test suite continues to serve as the main regression-guard in CI).

In our opinion, having a theorem prover in Smalltalk can be useful in a wider range of applications than our use-case of verified compilers.

At the time of writing, the implementation is still missing one piece, namely the “bounded” syntactic-sugar extension. This is important because it impedes the “optical” (aka “bidirectional”) expression of stateful load/store, central to reasoning about those very kinds of indeterminism we are most interested in. However, from research perspective this extension is exhaustively understood, therefore its implementation and testing introduces no technological uncertainty.

2.2. Tinyrossa

Tinyrossa (Vraný [47] and Vraný and Shingarov [18]) is a JIT- and AoT-compiler library based on traditional compilation algorithms. As far as *compilation* per se, we conservatively stay within well-established technology; the novelty of our contribution rests entirely on our formal treatment of those traditional algorithms. In particular, *Tinyrossa* reuses the OMR Testarossa Intermediate Language (TRIL), and to some degree follows Stoodley et al. [19] overall architecture.

Tinyrossa works by traversing 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).

In Stoodley et al. [19], this “non-TRIL” target means native CPU instructions (e.g. there are RISC-V evaluators, PowerPC evaluators *etc.*) The central idea of *Tinyrossa* is that every transformation – regardless of whether it is supplied by the compiler writer or found by synthesis – is endowed with a “Justification” (a certificate proving the transformation’s soundness). We achieve this by generalizing the “evaluator” concept to include “Semantic Evaluators”, whose target is the semantic core IR language of *MachineArithmetic* described in the previous chapter. That is, compiling for the Semantic target transforms a direct-style TRIL program into a CPS-style core-MA program, whose Floyd–Hoare axiomatic Verification Conditions (ultimately reduced to a Z3 SMT constraint) is obtained by Charguéraud [48] logical reflection (equivalently seen as Turchin supercompilation). This pipeline can be summarized like this:

TRIL → Core → Horn → Z3.

Justifying a TRIL-to-TRIL transformation then simply means checking the equivalence of the transformation’s source and target’s VC as a Z3 tautology. The entire operational semantics of each TRIL opcode is defined by its semantic evaluator.

Let us look at some examples, before turning to TRIL-to-Native transformations in Section 2.2.4.

2.2.1. Example: Gauss

First, let us look again at “Gauss” example, this time expressed in TRIL:

```
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>
```

In order to reason about the above TRIL program, we first need a formal semantics of TRIL opcodes. They are expressed here as functions in *MachineArithmetic* “semantic core” language (some of the used opcodes are omitted for brevity):

```
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)
};
```

The semantic evaluator then transforms the “Gauss” TRIL into two functions, one per each basic block:

```
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 trivial inlining, these two functions, combined with the above opcode-semantics functions (*iconst* *etc.*), are reduced to the sum “semantic core” program we saw in Section 2.1.1, where we explained how *MachineArithmetic* transforms an inductive proof of this program’s equivalence with $n * (n + 1) / 2$ into an SMT formula which is then discharged by Z3.

2.2.2. Example: If-compare simplification

As another example, of a different kind of TRIL-to-TRIL transformation, let us have a look at one particular optimization step (rule) called “if-compare simplification”. Its goal is to simplify code like `if ((a == b) == 1) ...` into simpler `if (a == b) ...`, removing the unnecessary comparison. Fig. 3 shows the TRIL tree (a) before and (b) after this optimization step.

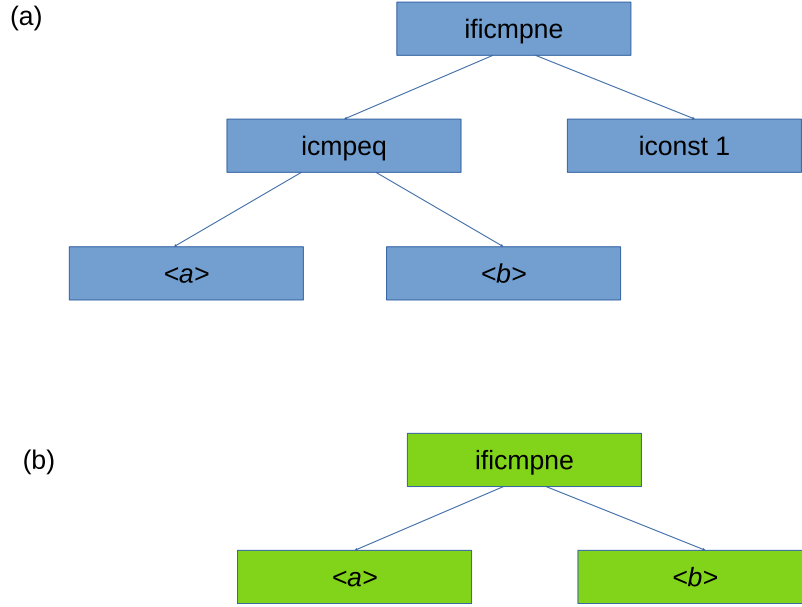


Fig. 3. If-compare simplification: (a) original tree, (b) simplified tree.

While it seems “obvious” that the simplified TRIL is equivalent to the original one (and therefore the optimization is correct), it ultimately depends on semantics of the `iconst`, `ificmpne` and `icmpeq` TRIL opcodes. Semantics of the first two has already been shown in Section 2.2.1, and `icmpeq` is:

```

reflect icmpeq : bv32 => bv32 => bv32
let icmpeq = (e1, e2) => {
  let eq = bveq32(e1, e2);
  if (eq) {
    bv32(1)
  } else {
    bv32(0)
  }
};

```

Each opcode semantics is modeled as function.

`iconst` is the simplest, it takes the constant as parameter and returns it.

`icmpeq` is similar, it takes two arguments (child nodes in TRIL) and returns 1 (as bitvector of length 32) if they are equal, 0 otherwise.

`ificmpne` is similar to `icmpeq` except it returns boolean value rather than 1 or 0. This value is then used to decide whether to take the branch or not.

First, the original TRIL fragment is rewritten as another function, taking arbitrary values `a` and `b` and two continuations `jump` and `cont` and returning either `jump` should the branch be taken or `cont` should it *not* be taken:

```

reflect original : bv32 => bv32 => 'b => 'b => 'b
let original = (a, b, jump, cont) => {
  let N001 = icmpeq(a, b);
  let N002 = iconst(bv32(1));
  let cond = ificmpne(vr3, vr4);
  if (cond) {
    jump
  } else {
    cont
  }
};

```

Similarly, the semantics of the simplified TRIL fragment is (after the “if-compare simplification” step):

```

reflect simplified : bv32 => bv32 => 'b => 'b => 'b
let simplified = (a, b, jump, cont) => {
  let cond = ificmpne(a, b);
  if (cond) {
    jump
  } else {
    cont
  }
};

```

Finally, to justify this particular transformation, we simply prove that for all values `a` and `b` and for any labels `l0` and `l1`, both functions `original` and `simplified` will return the same value — meaning in both cases execution would continue in the same direction (either `l0` and `l1`):

```

val static_assert : bool[b|b] => int
let static_assert = (b) => {
  0
};

val justify : int => int => 'b => 'b => int
let justify = (a, b, l0, l1) => {
  let orig = original (a, b, l0, l1);
  let simpl = simplified(a, b, l0, l1);
  let ok = orig == simpl;
  static_assert(ok)
};

```

2.2.3. Example: A look at a bug

Finally, let us look at the “absolute value” function:

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

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

For the integers, $\mathbb{Z}$, we express `abs` as

```

reflect abs : x:int => int

```

```
let abs = (x) => {
  let pos = x >= 0;
  if (pos) {
    x
  } else {
    0 - x
  }
};
```

and indeed

```
val main : int => bool[b|b]
let main = (y) => {
  let ya = abs(y);
  let ok = 0 <= ya;
  ok
};
```

typechecks.

We may be tempted to think that the same works for finite-precision machine integers, and write the following TRIL implementation of abs:

```
N000 bbstart <entry>
N016 ificmpge 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>
```

which semantic-evaluates down to:

```
reflect BB_002 : x:bv32 => bv32
let BB_002 = (x) => {
  let N010 = iload(x);
  ireturn(N010)
};
```

```
reflect abs : x:bv32 => bv32
let abs = (x) => {
  let N014 = iload(x);
  let N015 = iconst(bv32(0));
  let cond = ificmpge(N014, N015);
  if (cond) {
    BB_002(x)
  } else {
    let N004 = iconst(bv32(0));
    let N006 = isub(N004, N014);
    ireturn(N006)
  }
};
```

which, up to trivial function inlining, is equivalent to

```
reflect abs : x:bv32 => bv32
let abs = (x) => {
  let pos = x >= 0;
  if (pos) {
    x
  } else {
    0 - x
  }
};
```

Alas, the invariant does not typecheck:

```
val main : int => bool[b|b]
let main = (y) => {
  let ya = abs(y);
  let ok = bvle32(bv32(0), ya);
  ok
};
```

is Unsafe. Fortunately, the Unsafe object contains a Z3Model:

```
x = #x80000000
```

From this counterexample, we see what the problem is: #x80000000 is the 32-bit MININT. In two's complement, there are one less positive integers than negative ones, therefore abs(MININT) is too large to fit in a 32-bit word. 0-MININT underflows back into MININT.

2.2.4. ISA semantics

Justifying TRIL-to-Native transformations is much more involved. Existing work on mechanized reasoning about machine architecture, (such as CompCert [36], VST [37], CakeML [49], 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. [50] and Armstrong et al. [51] made progress towards 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,⁴ etc. Armstrong et al. [51]'s *Isla* tool transforms the authoritative full-scale ISA definition (such as ARM's ASL for Armv8-A, or the normative Sail definition of RISC-V) into their "Jib" intermediate language. Jib programs mutate memory by using the functions *read* and *write*, which in Jib itself are left uninterpreted: their definition – which characterizes the concurrency aspect of the memory model and is orthogonal to the ISA – is inferred from an axiomatic specification in Alglave et al. [52]'s "Cat" language.

Tinyrossa justifies TRIL-to-Native transformations ("evaluators") via abstract interpretation of Jib. We build this interpreter by reusing Sewell et al. [53]'s "Ott" tool. Fig. 4 shows the verification pipeline that transforms one single TRIL opcode's RISC-V evaluator into an SMT formula checked by Z3. Before this verification can take place, we preprocess the Sail specification of RISC-V ISA (1). First, the standard Sail compiler (2) emits the Jib IR (3); this step has nothing specific to Smalltalk-25, and happens outside of Smalltalk. To process this Jib IR, we build a Jib interpreter by feeding the Jib IL definition (4) into the external Ott tool (5) and piping the output (6) of a specialized "Smalltalk" Ott backend into a generic Ott-to-PetitParser translator-generator we call "PharoOtt" (10). The resulting translator takes the Jib IR as input and produces what we call an "ISA semantic function".

Once we have the RISC-V semantic function thus inferred from Sail, we can verify our evaluator. The evaluator (8) takes an Opcode (7) parametrized by some symbolic operands, and produces a "tile" of RISC-V code (9) which is parametrized by those operands. The application of the "RISC-V semantic function" to the tile produces what Armstrong et al. [51] call a "footprint", with one difference. Like other symbolic interpreters (such as *angr* etc.), Armstrong's footprint is a transformer from symbolic state to symbolic state, but is in itself fully concrete. In contrast, our footprint is a higher-order function over TRIL operands; in other words, the contents of instruction memory are themselves symbolic (Shingarov [10]).

⁴ Think of CMOIX scenarios such as a concurrent Polymorphic Inline Cache where one core updates the PIC while another core executes it.

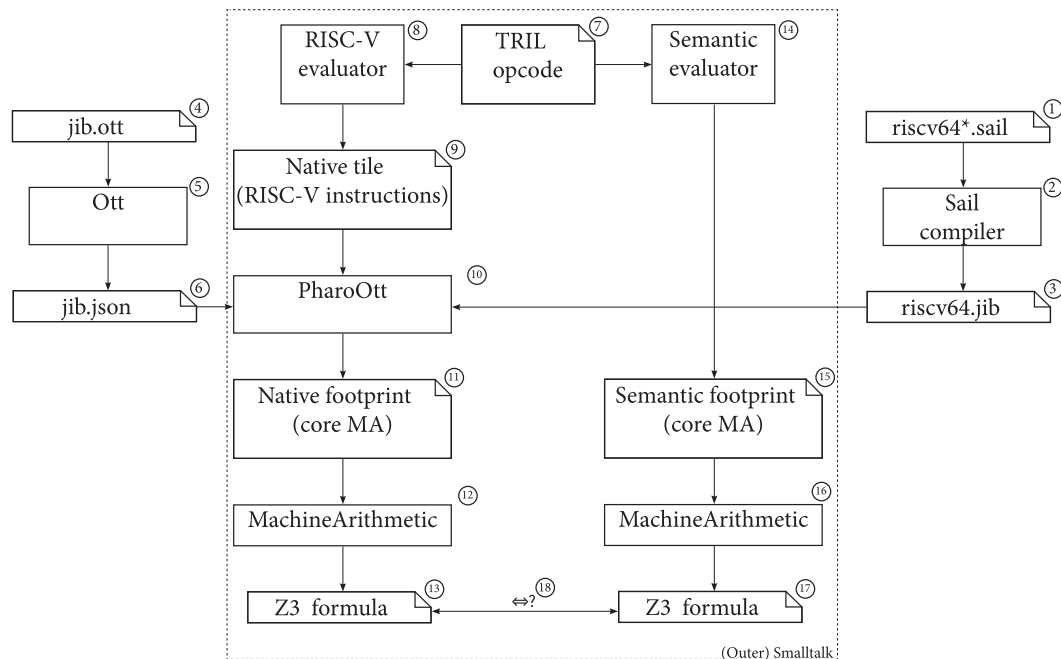


Fig. 4. Calculating the justification for an evaluator.

In *Tinyrossa*, the footprint (11) is a program in the core-language of *MachineArithmetic*. We have already seen how the supercompiler reflects it into SMT (12, 13) in order to check equivalence (18) with the output of the semantic evaluator (14–17).

2.2.5. Other sources of ISA semantics

Our target ISAs of most interest are RISC-V, OpenPOWER, ARM, and MIPS. One would like to hope that formal semantics specifications were available for these ISAs via some uniform way; unfortunately, this is not the case. To accommodate to this reality, we had to generalize our evaluator-justification framework to admit heterogeneous ISA specifications.

For example, even though a fragment of the POWER ISA has been formalized in Sail, the authoritative definition is the detailed pseudocode in the “Green Cloth” specification. Leighton [54] machine-processes this pseudocode to arrive at “ISACaller” semantics which they use for formal verification of the Libre-SOC OpenPOWER core.

Another interesting source of ISA semantics is the Valgrind VEX library used by Shoshitaishvili et al. [55]’s `angr` symbolic execution tools. Shingarov and Vraný [56] describe “VEX Regularization”, a process for interpolating an analytical form of instruction semantics from brute-force VEX-lifting all valid operand encodings.

2.2.6. Related work

At the initial stages of the project, we studied “direct” approaches to VM formalization.

Following Greve [57] and Liu and Moore [58] model the Java VM in ACL2 (Kaufmann et al. [59]).

Our early attempt to axiomatize Smalltalk using dependent types (Shingarov [43]) followed ideas in Atkey [60] who axiomatize the Java VM in the Coq proof assistant.

Barthe et al. [61] use Coq to machine-check their axiomatization of the JavaCard VM and BCV.

One limitation common to the above approaches is that these axiomatic models are to a greater or lesser degree idealized. To explore production VMs of interest to us, we traded scope for realism and (at least at this time) limited ourselves to the compiler-backend part of

the VM. Our chosen reference compiler, OMR Testarossa, is a traditional compiler; the literature on the verification algorithms for such traditional compilers is standard.

2.2.7. Current status and evaluation

Because the design and implementation of a compiler such as *Tinyrossa* is a large undertaking, one of *Tinyrossa*'s design goals was to allow modular evaluation. In other words, from the very beginning we wanted to be able to separate evaluating *Tinyrossa* as a *compiler* usable in a real Smalltalk, from evaluating its *formal verification*.

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 OpenPOWER and run them on real hardware, and in simulation (QEMU and gem5) for regression testing in CI.

The validation suite for the Justification component follows the general layout of LiquidHaskell’s benchmarks [62]: 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. The block that is still incomplete, is the PharoOtt interpreter. The current proof-of-concept PharoOtt runs on simple examples, but is not yet integrated with the Justification framework. Once this integration will be complete, so the semantic functions are automatically computed, it will allow us to (1) replace the hand-written fragment of TRIL semantics by a complete Ott TRIL specification, and (2) connect the Sail processing pipeline to the rest of *Tinyrossa*, and thus complete the automatic verification of the evaluators.

2.3. *PIG*

In the previous chapter we have established a method for modular verification of *Tinyrossa* transformation rules. The next step is to automatically *discover* such rules (in particular, native evaluators) by calculating them from the specification; thus-calculated evaluators are

correct by construction. *PIG*⁵ is a deductive synthesizer in the tradition of Manna and Waldinger [65].

To be of use to the practicing (JIT-)compiler-engineer, the synthesizer should present itself through an interface that uses familiar compiler-engineering concepts, such as e.g. of Massalin-tradition superoptimizer. Superoptimization is an approach to code generation where we start from a specification of a function and perform exhaustive search among instruction sequences. The seminal paper by Massalin [66] discusses the example of the *signum* function:

```
signum(x)
int x;
{
  if (x>0)      return 1;
  else if (x<0) return -1;
  else         return 0;
}
```

Most programmers would imagine an assembler implementation of *signum* in about eight or so instructions, including two conditional branches. Massalin's superoptimizer finds quirks in two's complement and carry, and does it in four 68020 instructions (no branches):

```
(x in d0)
addl d0,d0 | add d0 to itself
subxl d1,d1 | subtract (d1+Carry) from d1
negxl d0    | put (0-d0-Carry) into d0
addxl d1,d1 | add (d1+Carry) to d1
(signum(x) in d1)
```

Since that seminal 1987 paper, superoptimization has become a vast field of research. Keeping in view Smalltalk-25's goals, we do not attempt to advance the state of the art of superoptimization *per se*, — nor that of synthesis. Our approach has been to adopt a method that works fitting the design of Smalltalk-25. The characteristic of the latter which is most critical in the context of synthesis, is that verification based on refinement types *scales* because refinement-type-specifications *decompose*. In this setting, Manna–Waldinger deductive synthesis also scales. Polikarpova et al. [67] describe one such algorithm, which we implement in Smalltalk on top of *MachineArithmetic*.

One way to explain this algorithm, is as stepwise refinement [68]: to derive a program doing some work, we recursively perform top-down decomposition steps. At each step we delegate the work to some subprograms; or speaking game-theoretically, turn “our” work into “work for others”: an adversarial game between “our agent” vs. “the counteragent”. The program is thus decomposed into some subprograms (which could be – at the leaf level – atomic building blocks taken from a given “library of elementary functions”) and a formula combining them into the whole program.⁶ This is nothing more than combinatorics, plus the “well-typing” constraint that the actual and formal arguments' types should match. Because these are *refinement* types, they contain everything there is to say about the actual *values*. For example, 42 as defined unambiguously by its strongest type `Integer | [ : x | x==42 ]`. From this viewpoint, a function such as *increment*, which in the “coarse universe” looks something like

$$\begin{aligned} \text{increment} &:: \mathbb{Z} \rightarrow \mathbb{Z} \\ x &\mapsto x + 1, \end{aligned}$$

in the “refined universe”⁷ is a polymorphic *family* of functions, one function for each element of $\mathbb{Z}$:

$$\text{increment} :: \forall a \in \mathbb{Z}. \mathbb{Z} | [: x | x=a] \rightarrow \mathbb{Z} | [: y | y=(a+1)]$$

Taken together, these functions constitute the components of a natural transformation; de Bruin [72] shows that the naturality expresses exactly the “parametricity” condition, i.e. the absence of capturing or closing over some “secrets”. Parametricity is crucial to synthesis: it gives us “for free” certain parts of the specification without which the latter would be too weak but the explicit expression of which in the refinement would be too difficult; parametricity “fills in” our “canonical” expectations about “reasonable” behaviors.

As *MachineArithmetic* provides an explicit encoding for natural transformations and an explicit operation for checking naturality conditions (by reducing them to Z3 tautologies), expressing this kind of synthesis in our system is particularly simple.

Both the TRIL intermediate language and the native machine language are imperative, pointer-manipulating languages. Polikarpova and Sergey [73]’s “Synthetic Separation Logic of heap manipulation” updates the synthesis algorithm to address such situations; however, things become even more complicated when the building blocks cannot be regarded as primitive logical qualifiers, such as when synthesizing an instruction selector. A CPU instruction in any realistic ISA can perform a very complex sequence of operations. A Sail definition of an instruction can compile to many hundreds of Isla statements. We found this to be an impenetrable barrier to *realistic* fully automatic superoptimization. After several iterations of trying to implement *PIG* using various approaches, we currently prefer to give the compiler-backend engineer the flexibility of using *PIG* as a simple *assistant* in instruction-selector development, similar to proof assistants of the LCF tradition. Cases amenable to Polikarpova-like synthesis, are analogous to Coq’s “auto” tactic, while in more complex cases the machine helps by checking and recording the proof strategy. In the remaining aspects, this interactive engineering remains similar to the backwards-reasoning process of Polikarpova and Sergey [73]. Imagine we want to write 42 to location 0×1234 , on some RISC-like load/store architecture. Just as in Polikarpova and Sergey [73], we still match on the postcondition, which says that 0×1234 contains 42 and that the heap “outside of the frame” is left alone. To achieve this “Goal”, the compiler engineer applies the tactic “*stw*” (the “store word” instruction, made into a tactic by abstract interpretation of Isla). Now we have a new “Goal” (a temporary register must contain 42) and a “Justification”, the latter coming from typechecking the symbolic execution of the *stw* definition. In the next step, the engineer may apply a “load immediate” tactic (i.e. manually suggest a *li* instruction), thus concluding the proof.

2.3.1. Current status and evaluation

At the time of writing, *PIG* is the area of greatest technological uncertainty among the Smalltalk-25 components. During our Smalltalk-25 journey so far, we re-wrote this component several times over.

Our first attempt at a superoptimizer was the “Prolog Instruction Generator” which inferred the instruction selector from Auler et al. [74]’s AccGen model of ISA semantics. To evaluate this superoptimizer, we plugged it into Modtalk’s [75] CodeWriter API. We fed the AccGen model of PowerPC into it. The resulting *Target-Agnostic Modtalk* (Kilpela and Shingarov [14]) passes the “ANSITester” suite on the MPC5125 chip, which indicates that *PIG* does indeed emit meaningful PowerPC code. However, the performance is disappointing:

⁵ *PIG* originally stood for “Prolog Instruction Generator” to emphasize the influence of Green [63] and Waldinger and Lee [64]’s seminal resolution-theorem-proving-based synthesis on or work.

⁶ By Curry–Howard isomorphism [69], programs are proof terms for their specifications; a specification is a theorem whose premiss is the program’s input type and the conclusion is the output type. In this language, decomposition of programs corresponds to Milner’s decomposition of goals into subgoals [70]. We recognize that algorithms for goal-directed proof search are in fact program-derivation algorithms.

⁷ The reader familiar with the subject will recognize these are the target and source categories of the Melliès–Zeilberger forgetful functor [71].

Target-Agnostic Modtalk runs nine orders of magnitude slower than pure bytecode interpretation. The fundamental problem responsible for this extremely poor performance, is the weak expressive power of AccGen's RTL language, resulting in poor fidelity of the semantic model: in fact, the instructions most idiomatic to performant PowerPC code (such as `rlwinm`) are absent from the AccGen model altogether.

Our second-generation superoptimizer *Petrich* (Shingarov [9]) was based on Shoshitaishvili et al. [55]'s *angr* symbolic interpretation engine and the associated VEX ISA semantics, and operated directly at the level of Z3 API calls. Even though our proof-of-concept implementation was able to discover impressive optimizations in some isolated cases (such as Massalin's `sign()`), it never worked reliably, due to the now-well-known problem of brittle search heuristics in SMT solvers. Once we understood that directly calling SMT is too low-level to achieve reliable synthesis, we set out to implement the higher-level abstractions described in Sections 2.1 and 2.2. With these abstractions available, the solution to the synthesis problem is well-established in the literature.

Therefore, we see two pieces of unfinished work ahead of us. First, there is still that missing part of *MachineArithmetic* blocking us from reasoning about state. Once this gets implemented, we should be able to validate *PIG* against the standard benchmark of discovering all the tricks in "Hacker's Delight" (Warren [76]).

Second, there are certain "big-step" functional constructs specifically amenable to synthesis (Bird [77]). These turn out to be the same as the categorial constructs in terms of which we would like to reframe our Smalltalk (cf. Section 4).

2.4. Universally-Live Debugger

One of the most conspicuous characteristics of Smalltalk that makes it so attractive, is *Live Debugging*. Unfortunately, to use the live debugger, one needs a working system. This means that it is impossible to live-debug the low-level execution engine (the metaobject protocol, SmallIntegers, the VM, the CPU *etc*): we are fundamentally limited in that we cannot afford to break our existing toys. To overcome this, we extend *liveness* to *universal liveness*, where an "outer Smalltalk" (IDE running on the workstation) reflects on the execution state of "inner Smalltalk" (target). The principal idea that differentiates ULD from other "first-class interpreter" approaches such as Pinocchio (Verwaest et al. [78]), Metaphysics (Pimás and Marr [79]), or even VM simulation in Miranda et al. [80], is that ULD decouples the reflection layer from the mechanics of connecting to the target. Therefore, it does not dictate what physical nature the target can or cannot have.

At the time of writing, the ULD codebase offers a choice between two fully-implemented backends:

1. SmallRSP [3,5] is based on a low-level access API. Its use is advantageous when the underlying system provides no source-level debug facilities, but only the most primitive "absolute-level" operations (such as core stop/start, processor storage-base read/write, etc.). It contains a Smalltalk implementation of the GDB Remote Serial Protocol, which it uses by default, but offers a pluggable API for cases when even RSP is unavailable.
2. LibGDBs [7,8] is preferable when GDB is available. It communicates with GDB using the GDB/MI protocol. The scenario in which LibGDBs has the most advantage is when a production VM mixes [JIT- or AoT-]compiled Smalltalk code with C/C++ library code which has no reflection in the "first-class interpreter".

2.4.1. Current status and evaluation

At the time of writing, *ULD* is the most mature component in the Smalltalk-25 codebase; this is the debugger we use in our daily debug activities. A few interesting use-cases are worth describing.

In the *Target-Agnostic Cog* [12], we synthesize an OpenSmalltalk (Miranda et al. [80]) JIT for the POWER ISA. *TA-Cog* successfully executes Squeak's startup sequence (1729 message sends and 9000 byte-codes, taking 690 870 POWER instructions). The *TA-Cog* POWER evaluator is "admitted", i.e. simply trusted to be correct, instead of verified, because verification was not the principal focus of this experiment. The focus was on the use of *ULD* to improve observability of a VM as complex as OpenSmalltalk. Traditionally, OST VM can run either in "Simulation" (debugging a Smalltalk-like "Slang" notation in a Smalltalk debugger) or in "Production" (executing a Slang-to-C-transpiled program). The problem is that the simulated VM is a lax approximation of the production VM. We introduce a third, "SimProd" mode in which the production VM is reflected into the simulation environment. SimProd can be used from the traditional simulator GUI or from a new GT-based debugger offering three simultaneous perspectives of the running target:

- a CPU-native perspective, where the programmer can single-step JITted instructions, step-over Cog "abstract RTL", step-over C functions of the runtime, *etc*;
- a Smalltalk perspective of what the target VM is executing; and
- a JIT-postmortem "time-travel" perspective for observing how the now-executing native code was constructed by the JIT.

While we were debugging the Squeak startup sequence on POWER we realized the *ULD* offers an unexpected view of the substrate CPU. In particular (Shingarov [4]), this allowed us to find and fix a number of previously-unknown bugs in the gem5 (Lowe-Power et al. [81]) architecture simulator.

An extreme use-case for SmallRSP is debugging a broken processor. Shingarov [82] uses *ULD* to debug a port of the Microwatt [83] software implementation of OpenPOWER to the Xilinx Kintex FPGA.

Vraný [8] demonstrates the use of LibGDBs in a real-world scenario, debugging a crash in Smalltalk/X runtime. Smalltalk/X uses traditional VM implemented in C. Being able to see "logical" backtrace consisting of both Smalltalk method activations and calls to either VM or system libraries, together with ability to inspect Smalltalk objects on the heap helped to understand the nature of this particular crash.

Another interesting use of LibGDBs is debugging code execution in OpenJ9, a high-performance multi-threaded JVM implemented in C++. Certain parts, such as transitions from interpreted to compiled code or various "JIT helpers" are implemented in hand-written assembly. Once again, ability to maintain a "logical" backtrace across these transitions and to map JIT-compiled code back to the Java source proved to be invaluable.

As with much in Smalltalk-25, we have to make changes to upstream projects to enable our own progress. Changes which are deemed applicable to a wider range of uses than our Smalltalk-25 scenario are contributed back to respective upstream projects. In the case of LibGDBs, because it builds on top of GDB, supporting universally-live debugging required adding new Python API to GDB itself, to support JIT-compiled code and to support user-defined GDB/MI commands. It turned out this support is useful also outside of the context of debugging of dynamic-language systems.

In the future we plan to improve *ULD* in two ways.

Up to date, we have only used *ULD* to debug execution on fully-concrete targets. It will be interesting to plug our Isla-based symbolic interpreter into *ULD* as a third backend, along the lines of the proof-of-concept shown by Shingarov [9].

Another open question is how to communicate reflection from "outer Smalltalk" to GDB when using LibGDBs. To fully realize LibGDBs' main advantage – being able to reflect on execution of mixed (dynamic and C/C++) code in a seamless way – the GDB process on the side of the MI connection opposite to Smalltalk needs to know how to unwind stack, how to map native code back to source code, how

to traverse the object graph, etc.: the very same knowledge the first-class interpreter contains. One way to give this knowledge to GDB is to duplicate it in Python GDB scripts. This duplication is, of course, far from ideal. Another option would be to perform the reflection entirely on the Smalltalk side, but this would negate the benefit of LibDGBs, effectively turning it into SmallRSP. We want a GDB that could give us the best of both worlds: keep its knowledge about C/C++, while taking knowledge about Smalltalk from the authoritative first-class interpreter.

2.5. Ahead-of-Time

There is a long way between having a JIT-compiler and a running Smalltalk system. The authors' practical experience with engineering of production VMs suggests that the greatest peril along this way comes from taking the words "Virtual Machine" too literally. The concept of virtual machine is so attractive that one is easily lulled into *fully* believing its (partially-false) promise. Unfortunately, VMs are not truly virtual. A VM does not totally isolate the image from the details of the hardware; for example, the image code implementing the Integer hierarchy, concerns itself with the finite width of the underlying machine word. Nor does a VM isolate the image from the details of the VM's internal implementation; for example, most of the image code implementing `#become` is concerned with the internal VM tricks aimed at improving `#become`'s performance.

This is acceptable as long as we accept "approximately working" systems and "slightly leaking" abstraction barriers. Unfortunately, it is impossible to formally specify what it means for such "slightly broken" systems to be correct.

Multiple authors such as Feng [84] and Pimás et al. [85] propose to admit the problem with horizontally-layered technology stack, and replace broken abstraction barriers with useful integrations. We call this approach "vertical slicing".

To explore this vertical approach in the context of building a Smalltalk runtime, we combine the image and the VM into a homogeneous entity, following the design of Modtalk (Fridstrom et al. [75]) and Bee/LMR (Pimás et al. [85,86]). We begin from bootstrapping a non-live Smalltalk kernel. As this is a static program confined within a "code closure", its specification requires only traditional "static" methods. On the other hand, this kernel has enough functionality to "reintroduce liveness" to arrive at a full dynamic Smalltalk environment.

2.5.1. Related work

Fridstrom et al. [75] describe a research Smalltalk in which a "Program Definition" orchestrates the bootstrapping of "Subsystems" into a native program. Liveness is "reintroduced" into the system after it has bootstrapped.

Pimás et al. [86] and Pimás et al. [85] describe Bee/LMR, a "live metacircular runtime" for a production Smalltalk running a large industrial application. For our purposes, the AoT-bootstrapping of Bee is similar to that of Modtalk. The Bee Kernel roughly corresponds to Modtalk's ANSI Subsystem, the Code Closure to Program Definition, the Genesis to Object Writer, and the Nativizer to Code Writer.

2.5.2. Current status and evaluation

As a proof of this AoT-bootstrapping concept we implemented *Target-Agnostic Modtalk* (Kilpela and Shingarov [14]), an LMR for Modtalk (Fridstrom et al. [75]) AoT-compiled to PowerPC by our first-generation (refutation-based, AccGen-derived) *PIG*. Modtalk comes with a number of example "subsystems", including an "ANSI Smalltalk Subsystem" and the associated "ANSITester" suite to test for X3J20 compliance. *Target-Agnostic Modtalk* successfully runs this test, unmodified. Still, this experiment cuts two corners: (1) because *PIG v1* predates *Tinyrossa*, it does not use *Tinyrossa*'s verification system; and (2) several VM primitives (file I/O and other such functionality mandated by

ANSI but otherwise "boring") are implemented via "seminhosting", i.e. faulting back into the outer Smalltalk in *ULD* and emulating the primitive in the reflected interpreter.

To arrive at a usable system, we will change this proof-of-concept to connect a realistic LMR-bootstrappable Smalltalk to the *Tinyrossa* backend.

As we mentioned in Section 1, one of our goals is to extend Smalltalk beyond the "general-purpose computer" paradigm where the only structure available to represent objects are bytes in RAM. In our experiments so far we have only bootstrapped Smalltalk to ELF binaries, but in the future it will be interesting to target hardware such as SiliconSqueak (de Assumpção et al. [87] and de Assumpção [88]) in which objects are not necessarily structs of bytes and "pointer" is not necessarily synonymous with "address".

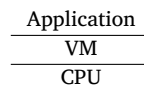
An open question to which we have no answer yet, is how to specify semantics after "reintroducing the liveness". The user is now allowed again to explore encapsulated objects by throwing messages at them and observing what happens. Does the exploratory nature of Smalltalk mean that the concept of "correct" vs. "incorrect" behavior does not exist? It will be interesting to come back to this question once we have a fully-running Smalltalk-25.

3. Lessons learned so far

These experiments seem to confirm our initial belief that it is possible to construct a Smalltalk environment for exploring formal reasoning about Smalltalk runtimes. At the very least, we did get the system to compute interesting answers. The insight gained from this exploration, caused a change in our view of not only how to implement Smalltalk, but also of diverse substrate technologies (GDB, gem5, OMR etc.) which led to upstream contributions that turned out to be valuable outside of Smalltalk.

Something that we did not anticipate, is that we could not build our system over a traditional horizontally-layered technology stack. We found several reasons for this. One reason is "technology bingo": even though all of the required building blocks may be available, they are implemented using incompatible technologies. For example, even though both the ISA-semantic framework of Armstrong et al. [50] and the Liquid Haskell theorem prover of Vazou [62] are both available and even based on the same calculus of Liquid Types, it is unclear how they can be integrated together.

In the last section we saw how the slightly-leaking horizontal "image/VM" abstraction barrier breaks the extremely attractive "Virtual Machine" promise. Kell [89] considers this to be a particular case of the more general problem with software developed in the traditional informal way, such as UNIX or C or the usual Smalltalk-80 implementations. Such software appears to work but only "almost". Under the extreme stress which a large-scale automated reasoning system puts on this software, it starts to break in subtle ways not manifest under "normal" circumstances. Worse, in a horizontally-layered technology stack, a foreign layer's problems tend to get worked-around in the layer affected by the problem (as opposed to the layer where the problem is). For example, in the following stack:



the Application may be affected by an issue that would be most appropriately addressed at the VM level. An Application-level workaround not only adds to the "approximate correctness" problem, it also makes *actually correct* work more difficult. It inevitably turns into exponentially increasing technical debt.

Feng [84] goes further than Pimás et al. [85] (who vertically slices the stack down to the VM) by slicing all the way down to the CPU. We see this approach as the way to go for Smalltalk-25, but this has implications for the language, which now must be the suitable language

for multiple layers, perhaps in a way similar to how Leighton [54] implements an OpenPOWER processor in Python.

4. Future work

In light of the project’s work-in-progress status, one desire that immediately comes to mind is to complete the building so we can have a formally-verified VM with all parts interacting together according to the grand plan. However, this is just the trivial aspect. The *interesting* aspect – which is the reason we regard Smalltalk-25 as a *Programm* as opposed to merely a *project* or even a *system* – is a return from Smalltalk-80 to an “early-spirit” Smalltalk, which our system will enable to bootstrap.

What is it about Smalltalk-80 that distinguishes it (in our view) from “original” Smalltalk?

The problem is that once designed, a Smalltalk-80 is set in stone: the design cannot be changed without breaking the system, fundamentally because its design objective is not to serve as a creative vehicle for this kind of exploration.

As we saw in Section 2.5, one such “immutable” design dimension is that a Smalltalk-80 VM is not “truly virtual”: instead of affording the image abstract structures to operate, the VM concerns the image with what the physical machine “is”: for example, SmallIntegers express the finite precision of the hardware’s integer arithmetic; the fixed implementation of Collection protocols reflects the uniprocessor nature of the machine; “object pointers” are memory addresses, following the prevailing “general-purpose computer” paradigm of that time; and so on.

The principal design goal of Smalltalk-25 has been to remove these hardcoded restrictions. The first step was the *Universally-Live Debugger*: we can now afford to break the experimental system. Reifying computation in *MachineArithmetic* is the second step. An indicative test of Smalltalk-25’s expressiveness – which we have not done yet – will be to attempt to use Smalltalk-25 as a more natural notational substrate to express the “Arrows” computer algebra system of Notarfrancesco [90]. An AoT-compiled Arrows – even though it is expressed over Smalltalk-80 concepts such as the Collection protocols which in the initial bootstrapping substrate are implemented with those Smalltalk-80 assumptions about sequential machine *etc.* – provides a running system that does not need to expose the initial hardcoded definition of Collections. For example, the definitional contract of `#collect:` is simply the functoriality condition, which is well-known to capture *just* enough to allow only the “correct” behavior(s) unambiguously (“no junk”) while at the same time not leak-in any irrelevant knowledge (“no noise”) e.g. about machine-representation details, because of the “universal property”.

It would be interesting to see what this “Smalltalk reimagined from a categorial point of view” – in which the basic departure points would be Functors, NaturalTransformations, SubobjectClassifiers *etc.* – would allow us to express.

CRedit authorship contribution statement

Boris Shingarov: Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Resources, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. **Jan Vraný:** Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Resources, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix. Glossary

API	— Application Programming Interface
AoT	— Ahead-of-Time compiler
ASL	— ARM Architecture Specification Language
AST	— Abstract Syntax Tree
CI	— Continuous Integration
CMODX	— Concurrent Modification and Execution
CPS	— Continuation Passing Style
DSL	— Domain Specific Language
ELF	— Executable and Linkable Format
FFI	— Foreign Function Interface
GDB	— The GNU Debugger
IL	— Intermediate Language
IR	— Intermediate Representation
ISA	— Instruction Set Architecture
JIT	— Just-in-Time compiler
LCF	— Logic for Computable Functions
LMR	— Live Metacircular Runtime
LQDT, Liquid	— Logically Qualified Data Type
ML	— Metalanguage
PIC	— Polymorphic Inline Cache
PIG	— Prolog Instruction Generator
RTL	— Register Transfer Language
SMT	— Satisfiability Modulo Theories
TRIL	— Testarossa Intermediate Language
ULD	— Universally-Live Debugger
VC	— Verification Condition
VM	— Virtual Machine

References

- [1] A. Turing, Checking a large routine, in: Report of a Conference on High Speed Automatic Calculating Machines, 1949, pp. 67–69.
- [2] B. Shingarov, The synthesis of target-agnostic JIT, in: 8th Smalltalks — Argentina Conference, Córdoba, Argentina, 2014.
- [3] B. Shingarov, Live introspection of target-agnostic JIT in simulation, in: Proceedings of the International Workshop on Smalltalk Technologies, IWST ’15, ACM, 2015.
- [4] B. Shingarov, Debugging a target-agnostic JIT compiler with gem5, in: ARM Research Summit, 2017.
- [5] B. Shingarov, Experiences with the universally-live debugger, in: 11th Smalltalks — Argentina Conference, La Plata, Argentina, 2017.
- [6] B. Shingarov, Dynamic language runtimes on RISC-V, in: 8th RISC-V Workshop Proceedings, 2018.
- [7] J. Vraný, LibGDBs – Smalltalk interface for GDB, 2014–2023, URL: <https://jan.vrany.io/hg/jv-libgdb/file/tip/README.md>.

- [8] J. Vraný, Using system level tools to debug Smalltalk systems, 2018, URL: <https://www.youtube.com/watch?v=Qbu4ED8wPYE>.
- [9] B. Shingarov, Executing the smalltalk VM algebraically, in: 13th Smalltalks — Argentina Conference, La Plata, Argentina, 2019.
- [10] B. Shingarov, Formal verification of JIT by symbolic execution, in: Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, in: VMIL 2019, 2019.
- [11] B. Shingarov, Colon stands for lambda: Generalized bound variables in Smalltalk, 2022, URL: <https://www.youtube.com/watch?v=r8RwJje0sOA>.
- [12] B. Shingarov, What is “debugging” anyway? Experiences with target-agnostic Cog, 2020, URL: <https://www.youtube.com/watch?v=kFni50dp9k>.
- [13] B. Shingarov, Towards a formally-verified software toolchain for open ISAs, 2020.
- [14] K. Kilpela, B. Shingarov, Target-agnostic modtalk, 2016.
- [15] B. Shingarov, Live proof by induction, in: FAST Workshop on Smalltalk Related Technologies, 2022.
- [16] J. Vraný, A year with Tinyrossa, 2023, URL: <https://www.youtube.com/watch?v=CVGQZtFM9L4>.
- [17] B. Shingarov, Towards a different Smalltalk, 2023, URL: <https://www.youtube.com/watch?v=PiFglgD8LNo>.
- [18] J. Vraný, B. Shingarov, Tinyrossa: a compiler framework for vertical, verified construction of Smalltalk VMs, in: 8th MoreVMs Workshop, in: Programming-2024, 2024.
- [19] M. Stoodley, et al., OMR, 2016–2024, URL: <https://github.com/eclipse/omr>.
- [20] T. Freeman, F. Pfenning, Refinement types for ML, SIGPLAN Not. 26 (6) (1991).
- [21] L. de Moura, N. Björner, Z3: An efficient SMT solver, in: C.R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 337–340.
- [22] B. Cosman, R. Jhala, Local refinement typing, Proc. ACM Program. Lang. 1 (ICFP) (2017).
- [23] N. Vazou, A. Tondwalkar, V. Choudhury, R.G. Scott, R.R. Newton, P. Wadler, R. Jhala, Refinement reflection: Complete verification with SMT, Proc. ACM Program. Lang. 2 (POPL) (2017).
- [24] N. Vazou, P. Rondon, R. Jhala, Abstract refinement types, in: Proceedings of the 22nd European Conference on Programming Languages and Systems, 2013.
- [25] N. Vazou, A. Bakst, R. Jhala, Bounded refinement types, in: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, in: ICFP 2015, 2015.
- [26] J.-C. Filliâtre, Proof of imperative programs in type theory, in: Types for Proofs and Programs, Springer, 1999.
- [27] J. Launchbury, S. Peyton Jones, State in Haskell, Lisp Symb. Comput. (1995).
- [28] J. McCarthy, A basis for a mathematical theory of computation, in: P. Braffort (Ed.), Computer Programming and Formal Systems, North-Holland, 1970.
- [29] R. Jhala, N. Vazou, Refinement types: A tutorial, Found. Trends Program. Lang. 6 (3–4) (2021) 159–317.
- [30] V. Turchin, The concept of a supercompiler, in: TOPLAS, 1986.
- [31] M.H. Sørensen, Turchin’s supercompiler revisited, 1996.
- [32] M. Bolingbroke, S. Peyton Jones, Supercompilation by evaluation, 2010.
- [33] E.W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Commun. ACM (1975).
- [34] M. Gordon, Edinburgh LCF: A mechanized logic of computation, 1980.
- [35] L.C. Paulson, Logic and computation: Interactive proof with cambridge LCF, 1987.
- [36] X. Leroy, Formal verification of a realistic compiler, Commun. ACM 52 (7) (July 2008) 107–115.
- [37] A.W. Appel, Program logics for certified compilers, 2014.
- [38] M. Barnett, R. Leino, Weakest-precondition of unstructured programs, in: PASTE ’05: The 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, 2005.
- [39] R. Leino, Dafny: An automatic program verifier for functional correctness, in: Logic for Programming, Artificial Intelligence, and Reasoning, 2010.
- [40] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, J. Yang, Secure distributed programming with value-dependent types, in: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP ’11, 2011.
- [41] R. Piskac, T. Wies, D. Zufferey, Automating separation logic using SMT, in: Computer Aided Verification, 2013.
- [42] C. Pulte, D.C. Makwana, T. Sewell, K. Memarian, P. Sewell, N. Krishnaswami, CN: Verifying systems C code with separation-logic refinement types, 2023.
- [43] B. Shingarov, Programming a smalltalk VM in coq, in: Proceedings of the International Workshop on Smalltalk Technologies, IWST ’17, ACM, 2017.
- [44] R. Leino, C. Pit-Claudel, Trigger selection strategies to stabilize program verifiers, in: Computer Aided Verification, 2016.
- [45] A. Tondwalkar, Finding and fixing bugs in liquid haskell, 2016.
- [46] W.T. Hallahan, A. Xue, M.T. Bland, R. Jhala, R. Piskac, Lazy counterfactual symbolic execution, in: PLDI 2019, 2019.
- [47] J. Vraný, Tinyrossa, 2022–2023, URL: <https://github.com/janvrany/Tinyrossa>.
- [48] A. Charguéraud, Program verification through characteristic formulae, 45 (2010) 321–332.
- [49] A. Fox, M.O. Myreen, Y.K. Tan, R. Kumar, Verified compilation of CakeML to multiple machine-code targets, in: Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, in: CPP 2017, 2017.
- [50] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K.E. Gray, R.M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, P. Sewell, ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS, in: Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages, 2019, Proc. ACM Program. Lang. 3, POPL, Article 71.
- [51] A. Armstrong, B. Campbell, B. Simmer, C. Pulte, P. Sewell, Isla: Integrating full-scale ISA semantics and axiomatic concurrency models (extended version), Form. Methods Syst. Des. (2023).
- [52] J. Alglave, P. Cousot, L. Maranget, Syntax and semantics of the weak consistency model specification language cat, 2016.
- [53] P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, R. Strniša, Ott: Effective tool support for the working semanticist, J. Funct. Programming (2010).
- [54] L. Leighton, The libre-SOC project, 2021, <https://ep2021.europython.eu/talks/46wiGSm-the-libre-soc-project/>.
- [55] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, G. Vigna, SOK: (state of) the art of war: Offensive techniques in binary analysis, in: 2016 IEEE Symposium on Security and Privacy, SP, 2016.
- [56] B. Shingarov, J. Vraný, VEX regularization, 2022.
- [57] D.A. Greve, Symbolic simulation of the JEM1 microprocessor, in: Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design, FMCAD ’98, 1998.
- [58] H. Liu, J.S. Moore, Executable JVM model for analytical reasoning: A study, in: Workshop on Interpreters, Virtual Machines and Emulators, ACM SIGPLAN, San Diego, California, USA, 2003.
- [59] M. Kaufmann, P. Manolios, J.S. Moore, Computer-Aided Reasoning: an Approach, Kluwer, 2000.
- [60] R. Atkey, CoqJVM: An executable specification of the java virtual machine using dependent types, in: Types for Proofs and Programs: International Conference, TYPES 2007, Cividale Des Friuli, Italy, May 2-5, 2007 Revised Selected Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 18–32.
- [61] G. Barthe, G. Dufay, L. Jakubiec, B.P. Serpette, S.M.d. Sousa, 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, UK, 2001, pp. 302–319, URL: <http://dl.acm.org/citation.cfm?id=645395.757559>.
- [62] N. Vazou, Liquid Haskell: Haskell as a Theorem Prover (Ph.D. thesis), University of California, San Diego, 2016.
- [63] C. Green, Application of theorem proving to problem solving, in: Proceedings of the 1st International Joint Conference on Artificial Intelligence, IJCAI ’69, 1969.
- [64] R.J. Waldinger, R.C.T. Lee, PROW: a step toward automatic program writing, in: Proceedings of the 1st International Joint Conference on Artificial Intelligence, IJCAI ’69, 1969.
- [65] Z. Manna, R. Waldinger, A deductive approach to program synthesis, ACM Trans. Program. Lang. Syst. (1980).
- [66] H. Massalin, Superoptimizer: A look at the smallest program, in: Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, in: ASPLOS II, Association for Computing Machinery, New York, NY, USA, 1987, pp. 122–126.
- [67] N. Polikarpova, I. Kuraj, A. Solar-Lezama, Program synthesis from polymorphic refinement types, in: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, in: PLDI, 2016.
- [68] R.-J. Back, J. von Wright, Refinement Calculus, Springer, 1998.
- [69] W.A. Howard, The formulae-as-types notion of construction, in: H. Curry, H. B., S.J. Roger, P. Jonathan (Eds.), To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism, Academic Press, 1980.
- [70] R. Milner, The use of machines to assist in rigorous proof, in: Proc. of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages, 1985.
- [71] P.-A. Mellies, N. Zeilberger, Functors are type refinement systems, in: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 3–16.
- [72] P.J. de Bruin, Naturalness of polymorphism, 1989.
- [73] N. Polikarpova, I. Sergey, Structuring the synthesis of heap-manipulating programs, Proc. ACM Program. Lang. (2019).
- [74] R. Auler, P.C. Centoducatte, E. Borin, ACGen: An automatic ArchC compiler generator, in: IEEE 24th International Symposium on Computer Architecture and High Performance Computing, IEEE, New York, NY, USA, 2012.
- [75] J. Fridstrom, A. Jacques, K. Kilpela, J. Sarkela, Testing modtalk, in: Lecture Notes in Business Information Processing, Agile Processes, in Software Engineering, and Extreme Programming — 16th International Conference, XP, Helsinki, Finland, 2015.

- [76] H.S. Warren, *Hacker's Delight*, second ed., Addison-Wesley Professional, 2012.
- [77] R.S. Bird, Algebraic identities for program calculation, *Comput. J.* (1989).
- [78] T. Verwaest, C. Bruni, D. Gurtner, A. Lienhard, O. Niestrasz, Pinocchio: bringing reflection to life with first-class interpreters, in: *OOPSLA '10*, 2010.
- [79] J. Pimás, S. Marr, *Metaphysics: Towards a robust framework for remotely working with potentially broken objects and runtimes*, 2017.
- [80] E. Miranda, C. Béra, E.G. Boix, D. Ingalls, Two decades of smalltalk VM development: Live VM development through simulation tools, in: *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, in: *VMIL 2018*, ACM, New York, NY, USA, 2018, pp. 57–66.
- [81] J. Lowe-Power, A.M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B.R. Bruce, D.R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S.A.R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T.M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L.E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M.D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D.A. Wood, H. Yoon, É.F. Zulian, *The gem5 simulator: Version 20.0+*, 2020.
- [82] B. Shingarov, *Live microhackathon: Toolchain support for Microwatt DMI*, 2023.
- [83] P. Mackerras, *Microwatt grows up*, 2021.
- [84] W. Feng, A vision for transforming 21st-century pedagogy via open standards: OpenPOWER, in: *OpenPOWER Summit 2021*, 2021.
- [85] J. Pimás, S. Marr, D. Garbervetsky, Live objects all the way down: Removing the barriers between applications and virtual machines, in: *The Art, Science, and Engineering of Programming*, Vol. 8, 2023.
- [86] J. Pimás, J. Burroni, G. Richarte, Design and implementation of Bee Smalltalk Runtime, in: *Proceedings of the International Workshop on Smalltalk Technologies, IWST '14*, ACM, 2014.
- [87] J.M. de Assumpção Jr., M. Voswinkel, E. Marques, Adapting processor grain via reconfiguration, in: *Reconfigurable Computing: Architectures, Tools, and Applications – 10th International Symposium, ARC 2014*, Vilamoura, Portugal, in: *Lecture Notes in Computer Science*, Vol. 8405, 2014.
- [88] J.M. de Assumpção Jr., Hardware/software codesign in neo smalltalk, in: *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '03*, 2003.
- [89] S. Kell, *Liberating the lurking Smalltalk in Unix*, 2014, URL: <https://www.youtube.com/watch?v=LwicN2u6Dro>.
- [90] L. Notarfrancesco, Arrows: a computer algebra system in smalltalk, in: *FAST Workshop 2022 on Smalltalk Related Technologies*, 2022.