Live Proof-by-Induction

Boris Shingarov *shingarov@labware.com*
LabWare

Abstract

Refinement types are a powerful formalism with many applications in program verification, compiler optimization, etc. We present our work-in-progress on MachineArithmetic, an implementation of refinement types in Smalltalk. Our current results are empirical, successfully running tests in the LiquidHaskell benchmark suite.

> They say you do not understand a topic in mathematics until you can teach it. And you do not really understand it until you can prove it in Coq. And you do not really *really* understand it until you can prove it in Agda.
>
> Russell O'Connor

> And you do not really *really REALLY* understand it until you have debugged why the proof fails.
>
> this author

1 Motivation

Designing an optimized Smalltalk Virtual Machine for direct execution on modern, post-Moore's-law processors presents new classes of challenges which processors with traditional sequentially-consistent ISAs did not pose. The lack of sequential consistency in the memory model not only breaks the assumptions of the common compiler optimizations (Morisset, 2017); the very approach where the systems programmer writes and tests code based on some intuitive argument why the code works, is — out of necessity — being replaced by programmatic manipulation of said argument (suitably formalized). In particular, when it comes to ISA semantics, we shift from relying exclusively on our informal understanding of the processor (which we gained by reading the processor manual, experimenting, falling into pit-traps etc.) towards machine-assisted analysis of formal specifications, e.g. using tools such as Sail (Armstrong et al., 2019; Wassell et al., 2018).

As Smalltalkers, we would of course prefer to have these tools homogeneous with our Smalltalk system therefore enjoying all the benefits of the Smalltalk paradigm (and even more so if we want to write the VM in Smalltalk). For this ambition, at a minimum we need to have a suitable automated-reasoning engine. This author's efforts in building such engine and such semantic tools, have led him to realize that in Smalltalk-80 our basic constructions lack certain desirable mathematical properties. In our arsenal of concepts, there is no structure that would satisfy the axioms of a topos; therefore we can't build a machine logic. Our classes do not compose; as well, they can be prone to expressing details of representation as opposed to intended meaning.

Let's consider a simple example.

LiveTyping (Wilkinson, 2019) aids the understanding of program behavior by collecting type information as the program runs. The nature of this information, is class membership. If I call the method

```
fourtyTwo
    ^42
```

a thousand times, from ten different call sites, LiveTyping will tell me that the return value was always a `SmallInteger`. How suitable is the granularity of this typing to my understanding of the program? is it too fine? too coarse?

Consider `Number>>abs`. If my program run contains the send: `42 negated abs` and the send: `42 factorial abs`, LiveTyping will report about `SmallInteger` and `LargePositiveInteger`. Here the question of granularity-relevance becomes especially evident (should `42` be a `SmallInteger`? a `Number`? a `PositiveInteger`?)[1] and the problem is not in LiveTyping but in the classes themselves: the fact that `42 abs` is a `SmallInteger`, is telling me something about how integer arithmetic is implemented on top of the underlying hardware, but not much about the meaning of `#abs`.

When I am reasoning about what `#abs` is supposed to return, at a minimum I expect some mention along the lines of `NonNegativeInteger` (to capture the "positivity" aspect) — something we do not have. Even if we did, we still can't compose it, for example, by intersecting with `SmallInteger` to obtain `SmallNonNegativeInteger` (standing for the integers that are both *small* and *nonnegative*): classes don't afford us logical connectives between propositions.

As another datapoint to make the emerging pattern more visible, let's consider the (nondestructive) function "add 1":

```
add1
    ^self + 1
```

If I throw a million different SmallIntegers at this function, I may (statistically very probably) get back a million SmallIntegers, leaving me with the impression that add1 **always** returns a SmallInteger. Such claim is both an underapproximation and an overapproximation. On the one hand, it is too weak to explain why we would reject

```
add1
    ^42
```

as an implementation for add1; on the other hand, it is too strong to be correct (i.e. SmallInteger not a superset of all possible values) becaues it misses the edge case where adding 1 overflows into LargeInteger. Thus our class-hierarchy of numbers starts to feel like the wrong level of abstraction to reason about what it means "to add 1 to an integer."

As a final illustrative example let's consider `Collection>>size`. If I have four tests, sending `#size` to `'big'`, `'pig'`, `'can'`, `'dig'`, in all cases the returned value is the same object `3`. What granularity would be appropriate to capture the essence of `#size`? Is it `SmallInteger`? `Number`? the particular object `3`? [2]

One could, of course, attempt to construct a class such as PositiveInteger, EvenInteger, etc. This immediately runs into some questions. How should these classes connect to other classes in the Number hierarchy? How do they connect to each other? What does "class membership" now mean? Is `42` an instance of PositiveInteger? EvenInteger? SmallInteger?

By all this, I don't mean to say that classes are broken or defective. The above few paragraphs try to delineate the task of creating a higher-level framework — which ideally would be embedded in our existing

[1] In a real-life debugging session, the complexity can easily derail the programmer's thinking so much they start asking questions as confused as "is the SmallInteger not PositiveInteger because of the negated?"

[2] Another dimension is that, size is polymorphic not only in its receiver, but also the receiver's elements; below we shall see why this is a major problem, and what we propose to do about it.

Smalltalk language as naturally as possible — suitable for building a computer logic in Smalltalk, which, in turn, would be suitable as a basis for automated reasoning about the Smalltalk VM along the lines of Morisset (2017) — Armstrong et al. (2019) – Wassell et al. (2018), or more generally, deriving the VM along the lines of the BMF formalism (Bird & de Moor, 1996). At a minimum, the objects in this framework need to have an algebra of logic — or in other words, satisfy the axioms of an (elementary) topos. Thus these type objects should correspond to their most-precise intended meaning (e.g. EvenInteger is exactly the set of even integers), compose (the result of sending

$PositiveInteger \cap EvenInteger$

is just another object of the same nature), have initial ($\emptyset$) and terminal ($\mathbb{Z}$) objects, etc. In such setting, the paradox with `42`'s membership does not arise because `42` is just the set of all $x \in \mathbb{Z}$ such that `x==42`.

This paper's task, then, is naturally a part of a hierarchy of tasks, at the top of which is the author's *Programm* aimed at verification and synthesis of Smalltalk VMs by formal methods.

2 Two naïve approaches

Before we are ready to explain the rationale for choosing the approach we propose in this paper, let's briefly consider why we failed when we tried two other, simple approaches.

2.1 First-order logic programming

Auler et al. (2012) synthesize a target-agnostic, tiling LLVM compiler backend from an extension to the ArchC PDL by using an ad-hoc unification procedure. We attempted to improve Auler's instruction selector for use in a Smalltalk JIT by systematically implementing Cheng et al. (1995)'s equational rewriter. This works well, even for very complex tiles, as long as the available instructions have only trivial "theory constraints", but fails at instructions such as PowerPC's `rlwinm`. In fact Auler's "CompilerInfo" for PowerPC does not attempt to correctly a model `rlwinm`'s semantics.

This is not a fault of Auler's implementation (or our attempted improvement in Smalltalk). The root cause of the problem is the nature of first-order logic programming as a free-term calculus.

Let's explain this by using a most extreme example.

Massalin's canonical example of his Superoptimizer (Massalin, 1987) automatically finds an instruction sequence implementing the `signum` function:

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

It would seem intuitively obvious that you need at least two comparisons and two jumps, but in reality you only need three register-arithmetic instructions; on SPARC:

```
    ! input in %i0, output in %o1
    addcc %i0 %i0 %l1
    subxcc %i0 %l1 %l2
    addx %l2 %i0 %o1
```

Massalin's astonishing trick works because of certain unexpected properties of the machine arithmetic of bit-vectors and flags — obscure / counterintuitive details of the behavior of carry and overflow in two's complement. These are rooted in *theory constraints*, as opposed to *free* structural properties. Logic programming easily unifies $(3 + 4)$ with $(X + Y)$, because this is term structure; but equating $3 + 4$ with 7 is part of the particular theory of integer arithmetic. It is not clear how to express the arithmetic properties

of two's complement carry in PROLOG or its extensions, so that an AccGen-like algorithm would discover Massalin's sequence or any of the idiomatic uses of `rlwinm`.

At this point, we started looking for something that could deal with rich theories and their combinations.

2.2 Direct use of SMT

Shoshitaishvili et al. (2016) use an SMT solver for static analysis of native machine binaries by symbolic execution. This appeared very attractive, for its faithful modeling of actual production ISAs. However, for our purposes this approach suffers from the problem opposite to what we described in section 2.1: while state-of-the-art SMT solvers excel at dealing with combinations of theories such as Linear Arithmetic, Bit Vectors, McCarthy arrays etc., they fundamentally are *SAT checkers* and are poor at *proving* in the PROLOG sense. In particular, they are *fragile*, i.e. prone to diverge under difficult-to-predict circomstances.

In the following discussion, we shall use the textbook example of the inductive `List` datatype, the simple problem of reasoning about the *length* function,

$length :: List\ 'a \rightarrow Integer$,

and our Smalltalk FFI to Microsoft Z3 (de Moura & Bjørner, 2008).

A List (of elements of some unspecified but fixed sort `'a`) can be thought of as an F-algebra comprising a coproduct of `Nil/0` and `Cons/2` recursively taking an `'a` and a List. This kind of inductive type declaration is directly supported by the Z3 API:

```
listType := Z3Datatype named: 'List'.
listType declare: 'cons' accessors: { 'car'->Int sort. 'cdr'->listType }.
listType declare: 'nil'.
listType := listType create.
```

Moreover, it is possible to declare the uninterpreted function *length*:

```
length := 'length' recursiveFunctionFrom: {listType} to: Int sort.
```

and assert inductive facts about its values:

```
length value: l is: (l is_nil ifTrue: [0 toInt] ifFalse: [(length value: l cdr) + 1]).
```

At this point Z3 can prove a few interesting facts about lists; for example: The length of a singleton is 1:

```
singleton := listType cons: x _: listType nil.
self assert: (
  Z3Solver isValid: (length value: singleton) === 1
)
```

Our Z3 FFI package contains a few other fun examples.

Let's try something more interesting. The function

$append :: List \times List \rightarrow List$

concatenates two lists by recursion:

```
append := 'append' recursiveFunctionFrom: {listType.listType} to: listType.
append valueWithArguments: {l.ll} is: (l is_nil
  ifTrue: [ll]
  ifFalse: [ | h t rest |
    h := l car.
    t := l cdr.
    rest := append value: t value: ll.
    listType cons: h _: rest ]).
```

We want to prove that the length of the "big" list is the sum of the lengths of the two "small" lists. To a human mathematician this is obvious by structural induction on the datatype, and indeed it is easy to formulate as a Z3 assertion:

```
xs := listType mkConst: 'xs'.
ys := listType mkConst: 'ys'.
self assert: (
  Z3Solver isValid:
    (length value: xs) + (length value: ys) "|xs|+|ys|"
      ===
    (length value: (append value: xs value: ys)) "|xs,ys|"
  )
```

Unfortunately, that last Z3 API call diverges in an infinite loop. Worse yet, it is almost unrealistic to predict when an e-match will diverge; so these capabilities of Z3 are unsuitable for use in automatic proof. There is a limited fragment of the logic which is guaranteed to be decidable; we are confined to it.

In the past decade, a number of algorithms for reducing inductive proof to the decidable fragment, has been studied extensively. Of course, in general this is not possible because of undecidability; so we are only dealing with "solvable cases" in the Ackermann (1954) sense.

One of these algorithms is Fusion described in Cosman & Jhala (2017) and used in their implementation of LQDT (Rondon et al., 2008). We studied Fusion and have implemented it in Smalltalk, *mutatis mutandis* due to our use of Smalltalk as opposed to Cosman–Jhala's Haskell implementation.

These differences are of three kinds.

- The nature of interfacing withe the SMT backend. Data at all stages along the transformation pipeline — the goal theorem, the Horn query, the tautology ultimately to be checked by Z3 — are live objects which can be inspected and manipulated in the debugger. This is in contrast to the batch-like SMT-LIB processing in LiquidFixpoint.

- In MachineArithmetic, the logical formulae serving as the refinement predicates are arbitrary Smalltalk code which compiles to opaque Blocks closing over transparent logical objects. In LiquidFixpoint predicates are *transparent* AST-tree terms processed by Visitor-like folding; reflecting arbitrary Haskell code back into refinements requires the additional technique of Proof by Logical Evaluation (PLE) (Vazou et al., 2017).

- Front-end interaction of MachineArithmetic with the host Smalltalk had to be designed from first principles, due to the difference between Smalltalk's and Haskell's programming paradigm.

Figure 1 shows the principal components of MachineArithmetic. The engine is the three layers in the middle; on either side is syntactic sugar.

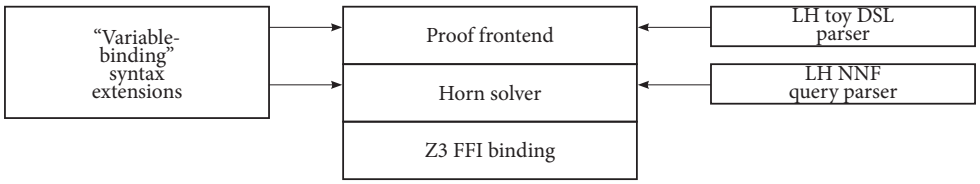


Figure 1: Principal components of MachineArithmetic.

3 Refinement types

In the communities of programming language design, optimizing compilers, theoretical computer science, logic, and many others, refinement types have amassed a large body of research and available literature since their introduction by Freeman & Pfenning (1991).

A refined type is built from a carrier set[3] (aka "unrefined type") and a predicate by "set comprehension". One can think of it as a subset of the carrier set: the predicate acts as the characteristic function of the subset, it is a block that takes an element of the carrier set and answers `true` if the element belongs to the subset, `false` if it's outside of the subset.

For example,

```
Integer | [ :x | x < 0 ]
```

is the set of all negative integers. The above is just normal Smalltalk; `#|` (pronounced "*such that*") is no different from any other binary selector[4].

Another simple example (let's call it σ_3 just for the next few pages) is the set of all three-character Strings:

```
String | [ :x | x size = 3 ]
```

Then $'big' \in \sigma_3$ and $'pig' \in \sigma_3$, but $'Smalltalk' \notin \sigma_3$.

One important special case of refinement is the singleton: i.e., the set $\{a\}$ given some particular/fixed element a. For example, the particular Integer, `42`, gives rise to the set

```
Integer | [ :x | x == 42 ].
```

This construction, which Ou et al. (2004) call "selfification", generalizes to "the most precise possible type for the term". The partial order of set inclusion (or, equivalently, implication between predicates) induces a topos of refinement types.

The author's inspiration to try using refinement types as the logical basis for his VM-verification/synthesis system, came from related work by (Armstrong et al., 2019; Wassell et al., 2018) on automated reasoning about optimizing-compiler correctness using formal specifications of ISAs and weakly consistent memory models based on the LQDT flavor of refinement types.

Why is it interesing to have refinement types in Smalltalk when there exist plenty of implementations in the ML tradition, besides serving as the foundation for the author's VM-synthesis *Programm*? To the author's best knowledge, *computing-of-proofs* has previously never been attempted in a Dynabook-like medium. By

[3]For the current exposition, we momentarily ignore the differentiation between sets and types despite its crucial importance; see Reynolds (1983) for more discussion. In our implementation, carriers are Sorts of the underlying SMT solver.

[4]There is nothing magical inside |, either: we just create a Refinement object holding the carrier and the block. These refinement objects get passed around, embedded in Forall quantifications, ANDed by $\wedge$, etc. until we can no longer avoid theory-specific operations — which happens at "UNSAT candidate checking" described at the end of section 4.1

"Dynabook-like", the author means that the cognitive activity of the human mathematician relies on a characteristic sensation of immediate contact with the mathematical construction. If a proof fails, the Smalltalk Debugger affords me to directly experience what's going on in the live computation. Diametrically opposite to this, up to date the *computing-of-proofs* has been dominated by the LCF/ML tradition, which explicitly declares the very term "*debugging* to be derogatory: "if you *need* to debug, then you are already doing it wrong".

In the author's opinion, the roots of this contradiction go back to Diophantus, who uses words such as "*four*" to refer ... [Do we need this?????] — a Dynabook with some kind of "intuitive / rigorous" switch or slider.

4 The algorithm

Conceptually, the algorithm we use is the Burstall–Darlington encoding (Burstall & Darlington, 1977). We refer to Cosman & Jhala (2017) for the description of details shared with their Haskell implementation; here we concentrate on the differences caused by the live nature of our Smalltalk implementation. The gist is as follows:

In step 1, the goal theorem is transformed into a Horn query in Negation Normal Form (NNF). This query is not first-order because it contains yet-unknown predicates (usually denoted κ).

The transformation is achieved by a repetitive, bidirectional "synthesize-check" cycle.

In step 2, the Horn query is reduced to the Burstall–Darlington representation by repeatedly eliminating unknown κ-variables using the Burstall–Darlington "Fold-Unfold" fixpoint algorithm.

In step 3, we pass the resulting κ-free tautology to Z3 for ckecking.

In counterbalance to the "rigorous" LCF/ML-tradition-style definitions of the algorithm — of which in the literature there are plenty — we illustrate by three simple examples. The first one, in section 4.1, verifies a straight-line (i.e. no loops/recursion) program (though it does contain conditional branches). The program in section 4.2 is even simpler; we include this example for extra clarity on what we said in section 2.1 about algebraic nonfreedom and combination of theories. Section 4.3 moves on to verify a recursive program by induction on datatype structure; this is where our principal variation from Cosman–Jhala happens.

4.1 Example 1: A straight-line program

Let's recall our `abs` example from section 1: in particular, the question about "how to capture the positivity aspect". Consider the following program in a toy programming language widely used for teaching in the LiquidHaskell community:

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

[[ val abs : x:int => int[?] ]]
let abs = (x) => {
  let pos = x >= 0;
  if (pos) {
    x
  } else {
    0 - x
  }
};

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

```
  assert(ok)
};
```

Here we see an implementation of `abs`'s body, which conditionally branches on whether the argument x is negative or not. We are attempting to prove that `abs`'s return value can never be negative. That is, we want a way to express this condition, and we want to build a checker that will be able to prove that our `abs` body satisfies it, by static analysis alone.

One straightforward approach would be to explicitly specify `abs`'s postcondition:

```
[[ val abs : x:int => int[ a | a >= 0 ] ]]
```

This is easy to implement and does work, but is a burden on the programmer and doesn't scale up to real programs. Instead, in the above program we leave the explicit contract of `abs` as unknown — we refer to it as a *hole* — and synthesize it at verification time from *asserting* that whatever we got back from `abs` is nonnegative.

Remarkably, `assert` is implemented entirely withing our toy language:

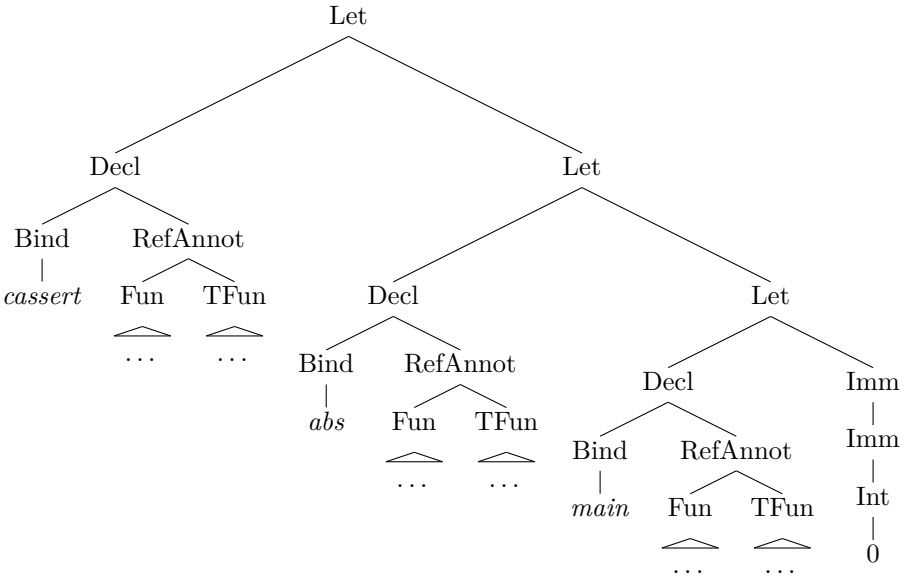
```
[[ val assert : bool[b|b] => int ]]
```

or perhaps it would be easier to understand if we write it, equivalently, as

```
[[ val assert : bool[b|b==true] => int ]]
```

The body of `assert` is irrelevant; only its domain ("all true Booleans") matters. The idea is that, for the call to `assert(ok)` on the last line of `main` to typecheck, we need to statically prove that ok cannot be anything other than "a true Boolean".

Parsing this program using PetitParser is trivial[5], resulting in an AST tree which we show here abridged for space's sake:



In the next phase of processing, we normalize this AST into Horn NNF form by recursive descent in a bidirectional check/synth cycle. We start with `#check:rtype:` of the root *Let* node.

`Let>>check:rtype:` is the Smalltalk implementation of the following Gentzen-style derivation rule:

[5]Of course this particular syntactic frontend is just an example, and the reason it exists in MachineArithmetic is for didactic and validation purposes.

$$\frac{\Gamma \vdash e \Rightarrow s \qquad \Gamma, x \in s \vdash e' \Leftarrow t'}{\Gamma \vdash \mathit{let}\ x = e\ \mathit{in}\ e' \Leftarrow t'}\ \text{[Chk-Let]}$$

That is, to check the conclusion at the bottom (the *Let*), we in turn call `#synth:` on the *Decl* (the left child in the above AST diagram) and `#check:rtype:` on the premiss *Let* (the right child).

Continuing this traversal recursively, applying rules similar to the "[Chk-Let]" above at each node, ultimately produces the following Horn VC:

$$\begin{array}{l}
\forall x \in \mathbb{Z} | \mathit{true}. \\
\quad \forall \mathit{pos} \in \mathbb{B} | \mathit{pos} \Leftrightarrow x \geq 0. \\
\quad \& \begin{cases} \forall g \quad \in \mathbb{B} | \mathit{pos}. \\ \qquad \forall v \in \mathbb{Z} | v = x. \quad \kappa(v, x) \\ \forall g \quad \in \mathbb{B} | \neg \mathit{pos}. \\ \qquad \forall v \in \mathbb{Z} | v = 0 - x. \quad \kappa(v, x) \end{cases}
\end{array}$$

(Here the precedence of the bar "|" and the dot "." is so that "$\forall x \in S | A.B$" reads "B holds for all x in S such that A".)

Here κ is the Horn unknown. It corresponds to the *hole* we talked about, and captures the meaning of `abs`: $\kappa(v, x)$ is a predicate taking an integer v and an integer x and returning `true` if v is the absolute value of x, `false` otherwise. But the system doesn't yet know κ; this is the work for the Horn solver, to synthesize it by solving the above NNF query. How does the Horn solver do this? If you take a formula such as $\forall x \in \mathbb{Z} | A.B$, κ may occur in it in the hypothesis (the A in $\forall x \in \mathbb{Z} | A.B$) or in the conclusion (the B). So these create a lower bound and an upper bound for κ. The remarkable thing is that these bounds determine κ uniquely (intuitively speaking, there is zero wiggle room in-between). Therefore we can unambiguously substitute the arguments to κ until all κ-unknowns are eliminated, leaving only Z3 tautologies (called "UNSAT candidates"). It is at this final stage that the predicates in refinements (the latter being built up in `#|`) become Z3 ASTs: at this point the "Alpha-uniquifier) has already renamed all potential name clashes, and we know the unrefined sort of every variable. We create leaf ASTs for each variable by calling `Z3_create_const()`, and put them into a new Smalltalk `Context`. Then the arbitrary Smalltalk blocks in the refinements constituting the UNSAT candidates, fire in that Context, returning a Z3 AST — it is that AST that we pass to Z3 to check.

4.2 Another straight-line program

Our next example is similar to the previous one. We find it illustrative because it explains our point about "theory-specific constraints". Here, we prove that the function `square` cannot ever return a negative value. For this, we are at the mercy of Z3's knowledge about the Integer Arithmetic theory. Theres is no structural information from which to deduce $x * x \geq 0$.

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

[[ val square : x:int => int[?] ]]
let square = (x) => {
  x * x
};

[[ val main : int => int ]]
let main = (x) => {
  let squared = square(x);
  let isPos = squared >= 0;
```

```
  assert(isPos)
};
```

This normalizes to the following Horn:

$$\begin{cases} \forall x \in \mathbb{Z} | true. \\ \quad \forall v \in \mathbb{Z} | v = (x * x). \kappa(v, x) \\ \forall y \in \mathbb{Z} | true. \\ \quad \forall squared \in \mathbb{Z} | \kappa(squared, y). \\ \quad\quad \forall isPos \in \mathbb{B} | isPos \Leftrightarrow squared \geq 0. \\ \quad\quad\quad \forall b \in \mathbb{B} | (b \Leftrightarrow squared \geq 0) \wedge (b = isPos). b \end{cases}$$

which is solved similarly to the previous section.

4.3 Inductive

Recall from section 2.2 the diverging attempted proof that the lengths of sub-lists add up to the length of the whole list. Here is the proof in our toy language:

```
[[ measure len : list('a) => int ]]

type list('a) =
  | Nil                      => [v| len v = 0]
  | Cons (x:'a, xs:list('a)) => [v| len v = 1 + len(xs)]
  ;

[[ val append : xs:list('a) => ys:list('a) => list('a)[v|len v = len(xs) + len(ys)] ]]
let rec append = (xs, ys) => {
  switch (xs) {
    | Nil        => ys
    | Cons(h, t) => let rest = append(t, ys);
                    Cons(h, rest)
  }
};
```

This needs surprisingly few new elements to normalize it to the same NNF form.

First, we see that `len` is a *measure*, a function that has no body and exists only as ghost code. In Z3 it instantiates to an *uninterpreted function*: nothing is known about `len` other than that it goes from some uninterpreted sort "(list 'a)" (which we shall discuss in the next section) to $\mathbb{Z}$, and that it returns equal values for equal arguments.

Normalizing the constructors "list = Nil | Cons" and deconstructors "switch = Nil | Cons" is straightforward.

Measure function application, such as `len v`, is trivial, amounting to a call to `Z3_mk_app()`.

4.4 The hotel of carrier sorts

A problem particular to our Smalltalk implementation, arises when a carrier type is parametrized. We saw an example of this when we wrote

length :: List 'a → Integer.

In general, such parametrization leads to a tree of carrier types representing the well-formedness of function-applications. In Cosman & Jhala (2017) this tree is built via unification of type terms by a pair of processes

called "monomorphisation" and "elaboration", which is possible because these type terms are just ASTs existing as regular Haskell data before the final tautology is passed to Z3. In Smalltalk this is not possible because carriers must exist as live objects pointing to actual Z3 sorts in Z3's memory. While it is possible to create the uninterpred sort `'a`, this breaks when we try to verify a call to a more-general function with a more-specific actual parameter sort: such as calling a function taking a list-of-anything with a list-of-integers, because `'a` is not a "supersort" of `Integer`. In other words, such uninterpreted sort `'a` is on the wrong side of "introduction rule" — expressing "*given a fixed a, ...*" instead of "*for all a, ...*". Following Reynolds (1983), Wadler (1989) and de Bruin (1989), we can say that

length :: List 'a $\rightarrow$ *Integer*

is a natural transformation: a family of morphisms *(List Integer)* $\rightarrow$ *Integer*, *(List Character)* $\rightarrow$ *Integer*, etc. In fact, this example is in a certain sense misleading, because the family is degenerate on the target side (the codomains of every morphism are all same). It would correspond to de Bruin's construction more closely if instead of Integer, each of these functions' return type would be something different depending on the argument type (aka, the morphism's domain). This immediately suggests to us the generalization stemming from the fact that the return value's refined type is dependent on the argument value, and then that values are types (by injection via Ou's "selfification").

How can we represent this natural transformation in Smalltalk? The first thing to worry about, is well-formedness with respect to matching the unrefined (carrier) types, because Z3 sorts don't unify so we can't construct a lattice of most-general-unifiers. The crucial observation is that no recursion is involved in the construction of these sort trees, so the total number of nodes is always finite. We place each sort node into a "hotel of sorts", which is an instance of class we call `IndexedSet`. This is a Smalltalk Set where each element is assigned an integer index, and this assignment does not change if the Set mutates. Imagine each node living in a numbered room in a hotel. Then we create a structureless, numbered uninterpreted Z3 sort, — unless we already unified the node with a sort we create previously, — one sort per room number. The second crucial observation is that this renaming of sorts does not change the outcome of the SMT check. This gives us our required representaion of "for all `'a`".

5 Some syntactic sugar

To allow natural notation for quantified expressions, Horn clauses, etc, we generalized Smalltalks's syntax for "dummy" bound variables (Shingarov, 2022).

In mathematics, "bound" (aka "dummy") variables are everywhere. Perhaps the first example of bound variables a teenager mathematician encounters, are integration variables: familiarizing with the idea that $\int_1^2 f(x)\,dx$ is the same as $\int_1^2 f(x)\,dx$, leading to the concept of α-equivalence. Further the student meets with λ-terms (equivalence classes of preterms modulo α), first-order quantifiers $\forall$ and $\exists$, higher-order Horn quantifier $\star$, and so on.

(...Transplant text from Shingarov-2022 abstract...)

We were careful to do it without increasing the complexity of existing syntax. In fact, the result is added conceptual clarity, because dummy-variable capture is now separated from the block-application semantics: a λ-block such as $[: x|x*2]$ can have "a value at $x = 5$ only because λ-abstractions have an "apply" operation attached to them, but in general simply capturing a dummy variable in a lexical unit is not enough to create such an operation; so it is meaningless to ask "what is the value of $[\forall x|x * 2 \geq 0]$ at $x = 5$." Our extension explicitly disentangles these two concepts.

In this notation, Smalltalk's nested lexical environments express type-dependency in an especially natural way: in

$[\forall x \in \mathbb{Z}|x * x \geq a]$

the closed-over a comes from the environment immediately outside of our block. This outer environment can also be a $\forall$-block, etc.

Thus, currently in MachineArithmetic there are three ways to create objects such as *Refinement, Forall, Exists* etc.

- Explicitly by sending an instantiation message (think `#new`) to the corresponding class.
- Using the above-described syntax.
- Using the parser for LiquidHaskell DSLs.

6 Results and Future work

This paper describes a work in progress. Currently we are reporting the following interim empirical result:

6.1 Experimental results

We implemented MachineArithmetic and as the first step in validating our implementation, verified that it passes tests from the LiquidHaskell benchmark suite (which we parse using the above-described technique). We chose this approach as fitting ideally with the spirit of Smalltalk and Dynabook, which affords effortless transition between intuition-building and precise abstract definitions.

6.2 Proof of the algorithm

In the next step, we intend to formally show that the theorems in Cosman & Jhala (2017) continue to hold in the live case.

6.3 Separation logic and Optics

Most importantly, out ultimate intent is to model memory and register access (Morisset, 2017; Armstrong et al., 2019; Wassell et al., 2018) as frames (McCarthy, 1970; McCarthy & Hayes, 1969) within our formalism, thus giving an axiomatization of the GET, PUT, STORE, LOAD operations in Shoshitaishvili et al. (2016).

6.4 Use outside the original motivation

An important goal for the future is to work with others towards applications other than our original motivation (compiler verification). In Alan Kay's words, Gutenberg's press becomes interesting the moment we apply it outside its original purpose of reproducing copies of the Bible **faster**. Being an implementation of LQDT, MachineArithmetic is a general intuitionistic reasoning engine. As such, for one possible "outside context" application, it would be interesting to use MachineArithmetic as the substrate type-checker for an LCF-style engine such as a Lean kernel, thus making the body of mathematics in the Xena project available in a Dynabook-like debuggable medium.

7 Availability

The complete source code is available from the author's GitHub page under MIT license.

References

W. Ackermann. *Solvable Cases of the Decision Problem*. North Holland, 1954.

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. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2019. doi: 10.1145/3290384. Proc. ACM Program. Lang. 3, POPL, Article 71.

Rafael Auler, Paulo Cesar Centoducatte, and Edson Borin. Accgen: An automatic archc compiler generator. *IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, 2012.

Richard Bird and Oege de Moor. *The Algebra of Programming*. Prentice-Hall, 1996. ISBN 013507245X. URL `http://www.cs.ox.ac.uk/publications/books/algebra/`.

R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, jan 1977. ISSN 0004-5411.

Mantis Cheng, Douglas Stott Jr Parker, and Maarten H. van Emden. A Method for Implementing Equational Theories as Logic Programs. In *Logic Programming: The 12th International Conference*. The MIT Press, 06 1995. ISBN 9780262291439. doi: 10.7551/mitpress/4298.003.0051. URL `https://doi.org/10.7551/mitpress/4298.003.0051`.

Benjamin Cosman and Ranjit Jhala. Local refinement typing. *Proc. ACM Program. Lang.*, 1(ICFP), aug 2017.

Peter J. de Bruin. Naturalness of polymorphism. 1989.

Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.

Tim Freeman and Frank Pfenning. Refinement types for ml. *SIGPLAN Not.*, 26(6), 1991. ISSN 0362-1340. doi: 10.1145/113446.113468. URL `https://doi.org/10.1145/113446.113468`.

Henry Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, pp. 122–126, New York, NY, USA, 1987. Association for Computing Machinery. ISBN 0818608056. doi: 10.1145/36206.36194. URL `https://doi.org/10.1145/36206.36194`.

John McCarthy. A basis for a mathematical theory of computation. In P. Braffort (ed.), *Computer programming and formal systems*. North-Holland, 1970.

John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie (eds.), *Machine Intelligence 4*, pp. 463–502. Edinburgh University Press, 1969.

Robin Morisset. *Compiler optimisations and relaxed memory consistency models*. Theses, Université Paris sciences et lettres, April 2017. URL `https://tel.archives-ouvertes.fr/tel-01823521`.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell (eds.), *Exploring New Frontiers of Theoretical Informatics*, pp. 437–450, Boston, MA, 2004. Springer US. ISBN 978-1-4020-8141-5.

John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pp. 513–523. North-Holland/IFIP, 1983.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pp. 159–169, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595938602. doi: 10.1145/1375581.1375602. URL `https://doi.org/10.1145/1375581.1375602`.

Boris Shingarov. Colon stands for lambda: Generalized bound variables in Smalltalk, 2022. URL `https://www.youtube.com/watch?v=r8RwJje0sOA`.

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, 2016. doi: 10.1109/SP.2016.17.

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: Complete verification with smt. *Proc. ACM Program. Lang.*, 2 (POPL), dec 2017. URL `https://doi.org/10.1145/3158141`.

Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pp. 347–359, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913280. doi: 10.1145/99370.99404. URL `https://doi.org/10.1145/99370.99404`.

Mark Wassell, Alasdair Armstrong, Neelakantan R. Krishnaswami, and Peter Sewell. Mechanised metatheory for the sail isa specification language. 2018.

Hernán Wilkinson. VM support for Live Typing: Automatic type annotation for dynamically typed languages. In *Programming '19*, 2019. ISBN 9781450362573.

Acronyms

AST Abstract Syntax Tree. 4

BMF Bird–Meertens Formalism. 2

ISA Instruction Set Architecture. 1

JIT Just-in-Time compiler. 2

LCF/ML Logic for Computable Functions / Meta Language. 6

LLVM Low Level Virtual Machine. 2

LQDT Logically Qualified Data Types, abbreviated to Liquid Types. 4, 9

NNF Negation Normal Form. 5

PDL Processor Description Language. 2

PLE Proof by Logical Evaluation. 4

SMT Satisfiability Modulo Theories. 3

VC Floyd–Hoare Verification Condition. 7

VM Virtual Machine. 1