Definitional Abstract Interpreters for Higher-Order Programming Languages

ANONYMOUS AUTHOR(S)

In this functional pearl, we examine the use of definitional interpreters as a basis for abstract interpretation of higher-order programming languages. As it turns out, definitional interpreters, especially those written in monadic style, can provide a nice basis for a wide variety of collecting semantics, abstract interpretations, symbolic executions, and their intermixings.

But the real insight of this story is a replaying of an insight from Reynold's landmark paper, *Definitional Interpreters for Higher-Order Programming Languages*, in which he observes definitional interpreters enable the defined-language to inherit properties of the defining-language. We show the same holds true for definitional *abstract* interpreters. Remarkably, we observe that abstract definitional interpreters can inherit the so-called "pushdown control flow" property, wherein function calls and returns are precisely matched in the abstract semantics, simply by virtue of the function call mechanism of the defining-language.

The first approaches to achieve this property for higher-order languages appeared within the last ten years, and have since been the subject of many papers. These approaches start from a state-machine semantics and uniformly involve significant technical engineering to recover the precision of pushdown control flow. In contrast, starting from a definitional interpreter, the pushdown control flow property is inherent in the meta-language and requires no further technical mechanism to achieve.

ACM Reference format:
Anonymous Author(s). 2017. Definitional Abstract Interpreters for Higher-Order Programming Languages. 1, 1, Article 1 (May 2017), 24 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

An abstract interpreter is intended to soundly and effectively compute an over-approximation to its concrete counterpart. For higher-order languages, these concrete interpreters tend to be formulated as state-machines (e.g. Suresh and Stephen (1995); Suresh et al. (1998); K. and Suresh (1998); Matthew and Olin (2006a); Jan and Thomas (2008); Jan and P. (2009); Matthew and David (2011); and Ilya et al. (2013)). There are several reasons for this choice: they operate with simple transfer functions defined over similarly simple data structures, they make explicit all aspects of the state of a computation, and computing fixed-points in the set of reachable states is straightforward. The essence of the state-machine based approach was distilled by Van Horn and Might in their "abstracting abstract machines" (AAM) technique, which provides a systematic method for constructing abstract interpreters from standard abstract machines like the CEK- or Krivine-machines (David and Matthew 2010). Language designers who would like to build abstract interpreters and program analysis tools for their language can now, in principle at least, first build a state-machine interpreter and then turn the crank to construct the approximating abstract counterpart.

A natural pair of questions that arise from this past work is to wonder:

XXXX-XXXX/2017/5-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(1) can a systematic abstraction technique similar to AAM be carried out for interpreters written, *not* as state-machines, but instead as high-level definitional interpreters, i.e. recursive, compositional evaluators?
(2) is such a perspective fruitful?

In this functional pearl, we seek to answer both questions in the affirmative.

For the first question, we show the AAM recipe can be applied to definitional interpreters in a straightforward adaptation of the original method. The primary technical challenge in this new setting is handling interpreter fixed-points in a way that is both sound and always terminates—a naive abstraction of fixed-points will be sound but isn't always terminating, and a naive use of caching for fixed-points will guarantee termination but is inherently unsound. We address this technical challenge with a straightforward caching fixed-point-finding algorithm which is both sound and guaranteed to terminate when abstracting arbitrary definitional interpreters.

For the second question, we claim that the abstract definitional interpreter perspective is fruitful in two regards. The first is unsurprising: high-level abstract interpreters offer the usual beneficial properties of their concrete counterparts in terms of being re-usable and extensible. In particular, we show that abstract interpreters can be structured with monad transformers to good effect. The second regard is more surprising, and we consider its observation to be the main contribution of this pearl.

Definitional interpreters, in contrast to abstract machines, can leave aspects of computation implicit, relying on the semantics of the defin*ing*-language to define the semantics of the defin*ed*-language, an observation made by Reynolds in his landmark paper, *Definitional Interpreters for Higher-order Programming Languages* (C. 1972). For example, Reynolds showed it is possible to write a definitional interpreter such that it defines a call-by-value language when the metalanguage is call-by-value, and defines a call-by-name language when the metalanguage is call-by-name. Inspired by Reynolds, we show that *abstract* definitional interpreters can likewise inherit properties of the metalanguage. In particular we construct an abstract definitional interpreter where there is no explicit representation of continuations or a call stack. Instead the interpreter is written in a straightforward recursive style, and the call stack is implicitly handled by the metalangauge. What emerges from this construction is a total abstract evaluation function that soundly approximates all possible concrete executions of a given program. But remarkably, since the abstract evaluator relies on the metalanguage to manage the call stack implicitly, it is easy to observe that it introduces no approximation in the matching of calls and returns, and therefore implements a "pushdown" analysis (Christopher et al. 2010; Dimitrios and Olin 2011), all without the need for any explicit machinery to do so.

Outline

In the remainder of this pearl, we present an adaptation of the AAM method to the setting of recursively-defined, compositional evaluation functions, a.k.a.∼definitional interpreters. We first briefly review the basic ingredients in the AAM recipe (section 2) and then define our definitional interpreter (section 3). The interpreter is largely standard, but is written in a monadic and extensible style, so as to be re-usable for various forms of semantics we examine. The AAM technique applies in a basically straightforward way by store-allocating bindings and soundly finitizing the heap. But when naively run, the interpreter will not always terminate. To solve this problem we introduce a caching strategy and a simple fixed-point computation to ensure the interpreter terminates (section 4). It is at this point that we observe the interpreter we have built enjoys the "pushdown" property *à la* Reynolds—it is inherited from the defining language of our interpreter and requires no explicit mechanism (section 5).

Having established the main results, we then explore some variations in brief vignettes that showcase the flexibility of our definitional abstract interpreter approach. First we consider the widely used technique of so-called "store-widening," which trades precision for efficiency by modelling the abstract store globally instead of locally (section 6). Thanks to our monadic formulation of the interpreter, this is achieved by a simple re-ordering of the monad transformer stack. We also explore some alternative abstractions, showing that due to the extensible construction, it's easy to experiment with alternative components for the abstract interpreter. In particular, we define an alternative interpretation of the primitive operations that remains completely precise until forced by joins in the store to introduce approximation (section 7). As another variation, we explore computing a form of symbolic execution as yet another instance of our interpreter (section 8). Lastly, we show how to incorporate so-called "abstract garbage collection," a well-known technique for improving the precision of abstract interpretation by clearing out unreachable store locations, thus avoiding future joins which cause imprecision (section 9). This last variation is significant because it demonstrates that even though we have no explicit representation of the stack, it is possible to compute analyses that typically require such explicit representations in order to calculate root sets for garbage collection.

Finally, we place our work in the context of the prior literature on higher-order abstract interpretation (section 10) and draw some conclusions (section 11).

Style

To convey the ideas of this paper as concretely as possible, we present code implementing our definitional abstract interpreter and all its variations. As a metalanguage, we use an applicative subset of Racket (Matthew and PLT 2010), a dialect of Scheme. This choice is largely immaterial: any functional language would do. However, to aide extensibility, we use Racket's *unit* system (Matthew and Matthias 1998) to write program components that can be linked together.

All of the code presented in this pearl runs; this document is a literate Racket program. We have also implemented a small DSL for composing and experimenting with these interpreters easily. Assuming Racket is installed, you can install the `monadic-eval` package with **(URL redacted for double-blind)** and a brief tutorial is available on github.

2 FROM MACHINES TO COMPOSITIONAL EVALUATORS

In recent years, there has been considerable effort in the systematic construction of abstract interpreters for higher-order languages using abstract machines—first-order transition systems—as a semantic basis. The so-called *Abstracting Abstract Machines* (AAM) approach to abstract interpretation (David and Matthew 2010) is a recipe for transforming a machine semantics into an easily abstractable form. The transformation includes the following ingredients:

- Allocating continuations in the store;
- Allocating variable bindings in the store;
- Using a store that maps addresses to *sets* of values;
- Interpreting store updates as a join; and
- Interpreting store dereference as a non-deterministic choice.

These transformations are semantics-preserving due to the original and derived machines operating in a lock-step correspondence. After transforming the semantics in this way, a *computable* abstract interpreter is achieved by:

- Bounding store allocation to a finite set of addresses; and
- Widening base values to some abstract domain.

After performing these transformations, the soundness and computability of the resulting abstract interpreter are then self-evident and easily proved.

The AAM approach has been applied to a wide variety of languages and applications, and given the success of the approach it's natural to wonder what is essential about its use of low-level machines. It is not at all clear

$e \in exp$	::=	(vbl x)	[*variable*]
	|	(num n)	[*conditional*]
	|	(if0 e e e)	[*binary op*]
	|	(app e e)	[*application*]
	|	(rec x ℓ e)	[*letrec*]
	|	ℓ	[*lambda*]
$\ell \in lam$	::=	(lam x e)	[*function defn*]
$x \in var$	::=	x, y, ...	[*variable name*]
$b \in bin$	::=	+, -, ...	[*binary prim*]

Figure 1: Programming Language Syntax

whether a similar approach is possible with a higher-level formulation of the semantics, such as a compositional evaluation function defined recursively over the syntax of expressions.

This paper shows that the essence of the AAM approach can be applied to a high-level semantic basis. We show that compositional evaluators written in monadic style can express similar abstractions to that of AAM, and like AAM, the design remains systematic. Moreover, we show that the high-level semantics offers a number of benefits not available to the machine model.

There is a rich body of work concerning tools and techniques for *extensible* interpreters (Javier 2009; Oleg 2012; Sheng et al. 1995), all of which applies to high-level semantics. By putting abstract interpretation for higher-order languages on a high-level semantic basis, we can bring these results to bear on the construction of extensible abstract interpreters.

3 A DEFINITIONAL INTERPRETER

We begin by constructing a definitional interpreter for a small but representative higher-order, functional language. The abstract syntax of the language is defined in Figure 1; it includes variables, numbers, binary operations on numbers, conditionals, `letrec` expressions, functions and applications.

The interpreter for the language is defined in Figure 2. At first glance, it has many conventional aspects: it is compositionally defined by structural recursion on the syntax of expressions; it represents function values as a closure data structure which pairs the lambda term with the evaluation environment; it is structured monadically and uses monad operations to interact with the environment and store; and it relies on a helper function δ to interpret primitive operations.

There are a few superficial aspects that deserve a quick note: environments ρ are finite maps and the syntax `(ρ x)` denotes $\rho(x)$ while `(ρ x a)` denotes $\rho[x \mapsto a]$. For simplicity, recursive function definitions (`rec`) are assumed to be syntactic values. The `do`-notation is just shorthand for `bind`, as usual:

```
(do x ← e . r) ≡ (bind e (λ (x) (do . r)))
     (do e . r) ≡ (bind e (λ (_) (do . r)))
 (do x ≔ e . r) ≡ (let ((x e)) (do . r))
         (do b) ≡ b
```

Finally, there are two unconventional aspects worth noting.

First, the interpreter is written in an *open recursive style*; the evaluator does not call itself recursively, instead it takes as an argument a function `ev`—shadowing the name of the function `ev` being defined—and `ev` (the argument) is called instead of self-recursion. This is a standard encoding for recursive functions in a setting without recursive binding. It is up to an external function, such as the Y-combinator, to close the recursive loop. This open recursive form is crucial because it allows intercepting recursive calls to perform "deep" instrumentation of the interpreter.

Second, the code is clearly *incomplete*. There are a number of free variables, typeset as italics, which implement the following:

- The underlying monad of the interpreter: *return* and *bind*;
- An interpretation of primitives: δ and *zero?*;
- Environment operations: *ask-env* for retrieving the environment and *local-env* for installing an environment;
- Store operations: *ext* for updating the store, and *find* for dereferencing locations; and
- An operation for *alloc*ating new store locations.

Going forward, we make frequent use of definitions involving free variables, and we call such a collection of such definitions a *component*. We assume components can be named (in this case, we've named the component `ev@`, indicated by the box in the upper-right corner) and linked together to eliminate free variables.[1]

Next we examine a set of components which complete the definitional interpreter, shown in Figure 3. The first component `monad@` uses a macro `define-monad` which generates a set of bindings based on a monad transformer stack. We use a failure monad to model divide-by-zero errors, a state monad to model the store, and a reader monad to model the environment. The `define-monad` form generates bindings for `return`, `bind`, `ask-env`, `local-env`, `get-store` and `update-store`; their definitions are standard (Sheng et al. 1995).

We also define `run` for running monadic computations, starting with the empty environment and store ∅:

```
(define (mrun m)
  (run-StateT ∅ (run-ReaderT ∅ m)))
```

While the `define-monad` form is hiding some details, this component could have equivalently been written out explicitly. For example, `return` and `bind` can be defined as:

```
(define (((return a) r) s) (cons a s))
(define (((bind ma f) r) s)
  (match ((ma r) s)
    [(cons a s′) (((f a) r) s′)]
    ['failure 'failure]))
```

So far our use of monad transformers is as a mere convenience, however the monad abstraction will become essential for easily deriving new analyses later on.

The `δ@` component defines the interpretation of primitives, which is given in terms of the underlying monad. The `alloc@` component provides a definition of `alloc`, which fetches the store and uses its size to return a fresh address, assuming the invariant `(∈ a σ)` ⇔ `(< a (size σ))`. The `alloc` function takes a single argument, which is the name of the variable whose binding is being allocated. For the time being, it is ignored, but will become relevant when abstracting closures (section 3.3). The `store@` component defines `find` and `ext` for finding and extending values in the store.

The only remaining pieces of the puzzle are a fixed-point combinator and the main entry-point for the interpreter, which are straightforward to define:

```
(define ((fix f) x) ((f (fix f)) x))
(define (eval e) (mrun ((fix ev) e)))
```

By taking advantage of Racket's languages-as-libraries features (Sam et al. 2011), we construct REPLs for interacting with this interpreter. Here are a few evaluation examples in a succinct concrete syntax:

[1]We use Racket *units* (Matthew and Matthias 1998) to model components in our implementation.

ev@

```
(define ((ev ev) e)
  (match e
    [(num n)
     (return n)]
    [(vbl x)
     (do ρ ← ask-env
         (find (ρ x)))]
    [(if0 e₀ e₁ e₂)
     (do v  ← (ev e₀)  z? ← (zero? v)
         (ev (if z? e₁ e₂)))]
    [(op2 o e₀ e₁)
     (do v₀ ← (ev e₀)  v₁ ← (ev e₁)
         (δ o v₀ v₁))]
    [(rec f l e)
     (do ρ  ← ask-env  a  ← (alloc f)
         ρ′ ≔ (ρ f a)
         (ext a (cons l ρ′))
         (local-env ρ′ (ev e)))]
    [(lam x e₀)
     (do ρ ← ask-env
         (return (cons (lam x e₀) ρ)))]
    [(app e₀ e₁)
     (do (cons (lam x e₂) ρ) ← (ev e₀)
          v₁ ← (ev e₁)
          a  ← (alloc x)
          (ext a v₁)
          (local-env (ρ x a) (ev e₂)))]))
```

Figure 2: The Extensible Definitional Interpreter

```
; Closure over the empty environment paired with the empty store.
> (λ (x) x)
'(((λ (x) x) . ()) . ())
; Closure over a non-empty environment and store.
> ((λ (x) (λ (y) x)) 4)
'(((λ (y) x) . ((x . 0))) . ((0 . 4)))
; Primitive operations work as expected.
> (* (+ 3 4) 9)
'(63 . ())
; Divide-by-zero errors result in failures.
> (quotient 5 (- 3 3))
'(failure . ())
```

```
                                                              monad@
(define-monad (ReaderT (FailT (StateT ID))))
                                                                  δ@
(define (δ o n₀ n₁)
  (match o
    ['+ (return (+ n₀ n₁))]
    ['- (return (- n₀ n₁))]
    ['* (return (* n₀ n₁))]
    ['/ (if (= 0 n₁) fail (return (/ n₀ n₁)))]))
(define (zero? v) (return (= 0 v)))
                                                              store@
(define (find a)  (do σ ← get-store
                      (return (σ a))))
(define (ext a v) (update-store (λ (σ) (σ a v))))
                                                              alloc@
(define (alloc x) (do σ ← get-store
                      (return (size σ))))
```

Figure 3: Components for Definitional Interpreters

```
                                                        trace-monad@
(define-monad (ReaderT (FailT (StateT (WriterT List ID)))))
                                                             ev-tell@
(define (((ev-tell ev₀) ev) e)
  (do ρ ← ask-env  σ ← get-store
      (tell (list e ρ σ))
      ((ev₀ ev) e)))
```

Figure 4: Trace Collecting Semantics

Because our monad stack places FailT above StateT, the answer includes the (empty) store at the point of the error. Had we changed monad@ to use (ReaderT (StateT (FailT ID))) then failures would not include the store:

```
> (quotient 5 (- 3 3))
'failure
```

At this point we've defined a simple definitional interpreter, although the extensible components involved—monadic operations and open recursion—will allow us to instantiate the same interpreter to achieve a wide range of useful abstract interpretations.

```
                                                        dead-monad@
(define-monad
  (ReaderT (StateT (StateT (FailT ID)))))
                                                           ev-dead@
(define (((ev-dead ev₀) ev) e)
  (do θ ← get-dead
      (put-dead (set-remove θ e))
      ((ev₀ ev) e)))
                                                         eval-dead@
(define ((eval-dead eval) e₀)
  (do (put-dead (subexps e₀))
      (eval e₀)))
```

Figure 5: Dead Code Collecting Semantics

3.1 Collecting Variations

The formal development of abstract interpretation often starts from a so-called "non-standard collecting semantics." A common form of collecting semantics is a trace semantics, which collects streams of states the interpreter reaches. Figure 4 shows the monad stack for a tracing interpreter and a "mix-in" for the evaluator. The monad stack adds `WriterT List`, which provides a new operation `tell` for writing lists of items to the stream of reached states. The `ev-tell` function is a wrapper around an underlying `ev₀` unfixed evaluator, and interposes itself between each recursive call by `tell`ing the current state of the evaluator: the current expression, environment and store. The top-level evaluation function is then:

```
(define (eval e) (mrun ((fix (ev-tell ev)) e)))
```

Now when an expression is evaluated, we get an answer and a list of all states seen by the evaluator, in the order in which they were seen. For example:

```
> (* (+ 3 4) 9)
'((63 . ())
  ((* (+ 3 4) 9) () ())
  ((+ 3 4) () ())
  (3 () ())
  (4 () ())
  (9 () ()))
```

Were we to swap `List` with `Set` in the monad stack, we would obtain a *reachable* state semantics, another common form of collecting semantics, that loses the order and repetition of states.

As another collecting semantics variant, we show how to collect the *dead code* in a program. Here we use a monad stack that has an additional state component (with operations named `put-dead` and `get-dead`) which stores the set of dead expressions. Initially this will contain all subexpressions of the program. As the interpreter evaluates expressions it will remove them from the dead set.

Figure 5 defines the monad stack for the dead code collecting semantics and the `ev-dead@` component, another mix-in for an `ev₀` evaluator to remove the given subexpression before recurring. Since computing the dead code requires an outer wrapper that sets the initial set of dead code to be all of the subexpressions in the program, we define `eval-dead@` which consumes a *closed evaluator*, i.e. something of the form `(fix ev)`. Putting these pieces together, the dead code collecting semantics is defined:

```
(define (eval e) (mrun ((eval-dead (fix (ev-dead ev))) e)))
```

Running a program with the dead code interpreter produces an answer and the set of expressions that were not evaluated during the running of a program:

```
> (if0 0 1 2)
(cons '(1 . ()) (set 2))
> (λ (x) x)
(cons '(((λ (x) x) . ()) . ()) (set 'x))
> (if0 (quotient 1 0) 2 3)
(cons '(failure . ()) (set 3 2))
```

Our setup makes it easy not only to express the concrete interpreter, but also these useful forms of collecting semantics.

3.2 Abstracting Base Values

Our interpreter must become decidable before it can be considered an analysis, and the first step towards decidability is to abstract the base types of the language to something finite. We do this for our number base type by introducing a new *abstract* number, written `'N`, which represents the set of all numbers. Abstract numbers are introduced by an alternative interpretation of primitive operations, given in Figure 6, which simply produces `'N` in all cases.

Some care must be taken in the abstraction of `'quotient`. If the denominator is the abstract number `'N`, then it is possible the program could fail as a result of divide-by-zero, since `0` is contained in the interpretation of `'N`. Therefore there are *two* possible answers when the denominator is `'N`: `'N` and `'failure`. Both answers are `return`ed by introducing non-determinism `NondetT` into the monad stack. Adding non-determinism provides the `mplus` operation for combining multiple answers. Non-determinism is also used in `zero?`, which returns both true and false on `'N`.

By linking together `δ^@` and the monad stack with non-determinism, we obtain an evaluator that produces a set of results:

```
> (* (+ 3 4) 9)
'((N . ()))
> (quotient 5 (+ 1 2))
'((failure . ()) (N . ()))
> (if0 (+ 1 0) 3 4)
'((4 . ()) (3 . ()))
```

It is clear that the interpreter will only ever see a finite set of numbers (including `'N`), but it's definitely not true that the interpreter halts on all inputs. First, it's still possible to generate an infinite number of closures. Second, there's no way for the interpreter to detect when it sees a loop. To make a terminating abstract interpreter requires tackling both. We look next at abstracting closures.

3.3 Abstracting Closures

Closures consist of code—a lambda term—and an environment—a finite map from variables to addresses. Since the set of lambda terms and variables is bounded by the program text, it suffices to finitize closures by finitizing

```
                                                              monad^@
(define-monad (ReaderT (FailT (StateT (NondetT ID)))))
                                                                   δ^@
(define (δ o n₀ n₁)
  (match* (o n₀ n₁)
    [('+ _ _)        (return 'N)]
    [('/ _ (? num?)) (if (= 0 n₁) fail (return 'N))]
    [('/ _ 'N)       (mplus fail (return 'N))] ...))
(define (zero? v)
  (match v
    ['N (mplus (return #t) (return #f))]
    [_  (return (= 0 v))]))
```

Figure 6: Abstracting Primitive Operations

```
                                                              alloc^@
(define (alloc x) (return x))
                                                            store-nd@
(define (find a)
  (do σ ← get-store
      (for/monad+ ([v (σ a)])
        (return v))))
(define (ext a v)
  (update-store (λ (σ) (σ a (if (∈ a σ) (set-add (σ a) v) (set v))))))
```

Figure 7: Abstracting Allocation: 0CFA

the set of addresses. Following the AAM approach, we do this by modifying the allocation function to produce elements drawn from a finite set. In order to retain soundness in the semantics, we modify the store to map addresses to *sets* of values, model store update as a join, and model dereference as a non-deterministic choice.

Any abstraction of the allocation function that produces a finite set will do, but the choice of abstraction will determine the precision of the resulting analysis. A simple choice is to allocate variables using the variable's name as its address. This gives a monomorphic, or 0CFA-like, abstraction.

Figure 7 shows the component `alloc^@` which implements monomorphic allocation, and the component `store-nd@` for implementing `find` and `ext` which interact with a store mapping to *sets* of values. The `for/monad+` form is a convenience for combining a set of computations with `mplus`, and is used so `find` returns *all* of the values in the store at a given address. The `ext` function joins whenever an address is already allocated, otherwise it maps the address to a singleton set. By linking these components with the same monad stack from before, we obtain an interpreter that loses precision whenever variables are bound to multiple values.

Our abstract interpreter now has a truly finite domain; the next step is to detect loops in the state-space to achieve termination.

4 CACHING AND FINDING FIXED-POINTS

At this point, the interpreter obtained by linking together `monad^@`, `δ^@`, `alloc^@` and `store-nd@` components will only ever visit a finite number of configurations for a given program. A configuration (ς) consists of an expression (`e`), environment (ρ) and store (σ). This configuration is finite because: expressions are finite in the given program; environments are maps from variables (again, finite in the program) to addresses; the addresses are finite thanks to `alloc^`; the store maps addresses to sets of values; base values are abstracted to a finite set by `δ^`; and closures consist of an expression and environment, which are both finite.

Although the interpreter will only ever see a finite set of inputs, it *doesn't know it.* A simple loop will cause the interpreter to diverge:

```
> (rec f (λ (x) (f x)) (f 0))
with-limit: out of time
```

To solve this problem, we introduce a *cache* (`$in`) as input to the algorithm, which maps from configurations (ς) to sets of value-and-store pairs (`v`$\times\sigma$). When a configuration is reached for the second time, rather than re-evaluating the expression and entering an infinite loop, the result is looked up from `$in`, which acts as an oracle. It is important that the cache is used co-inductively: it is only safe to use `in` as an oracle so long as some progress has been made first.

The results of evaluation are then stored in an output cache (`$out`), which after the end of evaluation is "more defined" than the input cache (`$in`), again following a co-inductive argument. The least fixed-point `$`$^{+}$ of an evaluator which transforms an oracle `$in` and outputs a more defined oracle $racket[out] is then a sound approximation of the program, because it over-approximates all finite unrollings of the unfixed evaluator.

The co-inductive caching algorithm is shown in Figure 8, along with the monad transformer stack `monad-cache@` which has two new components: `ReaderT` for the input cache `$in`, and `StateT+` for the output cache `$out`. We use a `StateT+` instead of `WriterT` monad transformer in the output cache so it can double as tracking the set of seen states. The `+` in `StateT+` signifies that caches for multiple non-deterministic branches will be merged automatically, producing a set of results and a single cache, rather than a set of results paired with individual caches.

In the algorithm, when a configuration ς is first encountered, we place an entry in the output cache mapping ς to `($in` ς`)`, which is the "oracle" result. Also, whenever we finish computing the result `v`$\times\sigma\prime$ of evaluating a configuration ς, we place an entry in the output cache mapping ς to `v`$\times\sigma\prime$. Finally, whenever we reach a configuration ς for which a mapping in the output cache exists, we use it immediately, `return`ing each result using the `for/monad+` iterator. Therefore, every "cache hit" on `$out` is in one of two possible states: 1) we have already seen the configuration, and the result is the oracle result, as desired; or 2) we have already computed the "improved" result (w.r.t. the oracle), and need not recompute it.

To compute the least fixed-point `$`$^{+}$ for the evaluator `ev-cache` we perform a standard Kleene fixed-point iteration starting from the empty map, the bottom element for the cache, as shown in Figure 9.

The algorithm runs the caching evaluator `eval` on the given program `e` from the initial environment and store. This is done inside of `mlfp`, a monadic least fixed-point finder. After finding the least fixed-point, the final values and store for the initial configuration ς are extracted and returned.

Termination of the least fixed-point is justified by the monotonicity of the evaluator (it always returns an "improved" oracle), and the finite domain of the cache, which maps abstract configurations to pairs of values and stores, all of which are finite.

With these pieces in place we construct a complete interpreter:

```
(define (eval e) (mrun ((fix-cache (fix (ev-cache ev))) e)))
```

monad-cache@

```
(define-monad (ReaderT (FailT (StateT (NondetT (ReaderT (StateT+ ID)))))))
```

ev-cache@

```
(define (((ev-cache ev₀) ev) e)
  (do ρ    ← ask-env  σ ← get-store
      ς    ≔ (list e ρ σ)
      $⸢out⸣ ← get-cache-out
      (if (∈ ς $⸢out⸣)
          (for/monad+ ([v×σ ($⸢out⸣ ς)])
            (do (put-store (cdr v×σ))
                (return (car v×σ))))
          (do $⸢in⸣    ← ask-cache-in
              v×σ₀  ≔ (if (∈ ς $⸢in⸣) ($⸢in⸣ ς) ∅)
              (put-cache-out ($⸢out⸣ ς v×σ₀))
              v     ← ((ev₀ ev) e)
              σ′    ← get-store
              v×σ′  ≔ (cons v σ′)
              (update-cache-out (λ ($⸢out⸣) ($⸢out⸣ ς (set-
add ($⸢out⸣ ς) v×σ′))))
              (return v)))))
```

Figure 8: Co-inductive Caching Algorithm

fix-cache@

```
(define ((fix-cache eval) e)
  (do ρ ← ask-env  σ ← get-store
      ς ≔ (list e ρ σ)
      $⁺ ← (mlfp (λ ($) (do (put-cache-out ∅)
                             (put-store σ)
                             (local-cache-in $ (eval e))
                             get-cache-out)))
      (for/monad+ ([v×σ ($⁺ ς)])
        (do (put-store (cdr v×σ))
            (return (car v×σ))))))
(define (mlfp f)
  (let loop ([x ∅])
    (do x′ ← (f x)
        (if (equal? x′ x) (return x) (loop x′)))))
```

Figure 9: Finding Fixed-Points in the Cache

When linked with `δ^` and `alloc^`, this abstract interpreter is sound and computable, as demonstrated on the following examples:

```
> (rec f (λ (x) (f x))
    (f 0))
'()
> (rec f (λ (n) (if0 n 1 (* n (f (- n 1)))))
    (f 5))
'(N)
> (rec f (λ (x) (if0 x 0 (if0 (f (- x 1)) 2 3)))
    (f (+ 1 0)))
'(3 2 0)
```

Formal soundness and termination

In this pearl, we have focused on the code and its intuitions rather than rigorously establishing the usual formal properties of our abstract interpreter, but this is just a matter of presentation: the interpreter is indeed proven sound and computable. We have formalized

this co-inductive caching algorithm in Appendix A, where we prove both that it always terminates, and that it computes a sound over-approximation of concrete evaluation. Here, we give a short summary of our metatheory approach.

In formalising the soundness of this caching algorithm, we extend a standard big-step evaluation semantics into a *big-step reachability* semantics, which characterizes all intermediate configurations which are seen between the evaluation of a single expression and its eventual result. These two notions—*evaluation* which relates expressions to fully evaluated results, and *reachability* which characterizes intermediate configuration states—remain distinct throughout the formalism.

After specifying evaluation and reachability for concrete evaluation, we develop a *collecting* semantics which gives a precise specification for any abstract interpreter, and an *abstract* semantics which partially specifies a sound, over-approximating algorithm w.r.t. the collecting semantics.

The final step is to compute an oracle for the *abstract evaluation relation*, which maps individual configurations to abstractions of the values they evaluate to. To construct this cache, we *mutually* compute the least-fixed point of both the evaluation and reachability relations: based on what is evaluated, discover new things which are reachable, and based on what is reachable, discover new results of evaluation. The caching algorithm developed in this section is a slightly more efficient strategy for solving the mutual fixed-point, by taking a deep exploration of the reachability relation (up-to seeing the same configuration twice) rather than applying just a single rule of inference.

5 PUSHDOWN *À LA* REYNOLDS

By combining the finite abstraction of base values and closures with the termination-guaranteeing cache-based fixed-point algorithm, we have obtained a terminating abstract interpreter. But what kind of abstract interpretation did we get? We have followed the basic recipe of AAM, but adapted to a compositional evaluator instead of an abstract machine. However, we did manage to skip over one of the key steps in the AAM method: we never store-allocated continuations. *In fact, there are no continuations at all!*

A traditional abstract machine formulation of the semantics would model the object-level stack explicitly as an inductively defined data structure. Because stacks may be arbitrarily large, they must be finitized like base values and closures, and like closures, the AAM trick is to thread them through the store, which itself must become finite. But in the definitional interpreter approach, the story of this paper, the model of the stack is implicit and simply inherited from the meta-language.

But here is the remarkable thing: since the stack is inherited from the meta-language, the abstract interpreter inherits the "call-return matching" of the meta-language, which is to say there is no loss of precision of in the analysis of the control stack. This is a property that usually comes at considerable effort and engineering in the formulations of higher-order flow analysis that model the stack explicitly. So-called higher-order "pushdown" analysis has been the subject of multiple publications and two dissertations (Christopher et al. 2012; Christopher et al. 2010; David and Matthew 2012; Dimitrios and Olin 2011; Earl 2014; Ian et al. 2014; J.∼Ian and David 2014; Thomas et al. 2016; Vardoulakis 2012). Yet when formulated in the definitional interpreter style, the pushdown property requires no mechanics and is simply inherited from the meta-language.

Reynolds, in his celebrated paper *Definitional Interpreters for Higher-order Programming Languages* (C. 1972), first observed that when the semantics of a programming language is presented as a definitional interpreter, the defined language could inherit semantic properties of the defining metalanguage. We have now shown this observation can be extended to *abstract* interpretation as well, namely in the important case of the pushdown property.

In the remainder of this paper, we explore a few natural extensions and variations on the basic pushdown abstract interpreter we have established up to this point.

6 WIDENING THE STORE

In this section, we show how to recover the well-known technique of store-widening in our formulation of a definitional abstract interpreter. This example demonstrates the ease of which we can construct existing abstraction choices.

The abstract interpreter we've constructed so far uses a store-per-program-state abstraction, which is precise but prohibitively expensive. A common technique to combat this cost is to use a global "widened" store (Matthew 2007a; Olin 1991), which over-approximates each individual store in the current set-up. This change is achieved easily in the monadic setup by re-ordering the monad stack, a technique due to David et al. (2015). Whereas before we had `monad-cache@` we instead swap the order of `StateT` for the store and `NondetT`:

```
(ReaderT (FailT (NondetT (StateT+ (ReaderT (StateT+ ID))))))
```

we get a store-widened variant of the abstract interpreter. Because `StateT` for the store appears underneath nondeterminism, it will be automatically widened. We write `StateT+` to signify that the cell of state supports such widening.

7 AN ALTERNATIVE ABSTRACTION

In this section, we demonstrate how easy it is to experiment with alternative abstraction strategies by swapping out components. In particular we look at an alternative abstraction of primitive operations and store joins that results in an abstraction that—to the best of our knowledge—has not been explored in the literature. This example shows the potential for rapidly prototyping novel abstractions using our approach.

Figure 10 defines two new components: `precise-δ@` and `store-crush@`. The first is an alternative interpretation for primitive operations that is *precision preserving*. Unlike `δ^@`, it does not introduce abstraction, it merely propagates it. When two concrete numbers are added together, the result will be a concrete number, but if either number is abstract then the result is abstract.

This interpretation of primitive operations clearly doesn't impose a finite abstraction on its own, because the state space for concrete numbers is infinite. If `precise-δ@` is linked with the `store-nd@` implementation of the store, termination is therefore not guaranteed.

```
                                                              precise-δ@
(define (δ o n₀ n₁)
  (match* (o n₀ n₁)
    [('+ (? num?) (? num?)) (return (+ n₀ n₁))]
    [('+ _        _)        (return 'N)] ...))
(define (zero? v)
  (match v
    ['N (mplus (return #t) (return #f))]
    [_  (return (= 0 v))]))
```

```
                                                              store-crush@
(define (find a)
  (do σ ← get-store
      (for/monad+ ([v (σ a)])
        (return v))))
(define (crush v vs)
  (if (closure? v)
      (set-add vs v)
      (set-add (set-filter closure? vs) 'N)))
(define (ext a v)
  (update-store (λ (σ) (if (∈ a σ)
                           (σ a (crush v (σ a)))
                           (σ a (set v))))))
```

Figure 10: An Alternative Abstraction for Precise Primitives

The `store-crush@` operations are designed to work with `precise-δ@` by performing *widening* when joining multiple concrete values into the store. This abstraction offers a high-level of precision; for example:

```
; Constant arithmetic expressions are computed with full precision.
> (* (+ 3 4) 9)
'(63)
; Even linear binding and arithmetic preserves precision.
> ((λ (x) (* x x)) 5)
'(25)
; Only when the approximation of binding structure comes in to
; contact with base values that we see a loss in precision.
> (let f (λ (x) x)
    (* (f 5) (f 5)))
'(N)
```

This combination of `precise-δ@` and `store-crush@` allows termination for most programs, but still not all. In the following example, `id` is eventually applied to a widened argument `'N`, which makes both conditional branches reachable. The function returns `0` in the base case, which is propagated to the recursive call and added to `1`, which yields the concrete answer `1`. This results in a cycle where the intermediate sum returns `2`, `3`, `4` when applied to `1`, `2`, `3`, etc.

```
                                                                    symbolic-monad@
 (define-monad (ReaderT (FailT (StateT (StateT (NondetT ID))))))
                                                                      ev-symbolic@
 (define (((ev-symbolic ev₀) ev) e)
   (match e [(sym x) (return x)]
            [e       ((ev₀ ev) e)]))
                                                                       δ-symbolic@
 (define (δ o n₀ n₁)
   (match* (o n₀ n₁)
     [('/ n₀ n₁) (do z? ← (zero? n₁)
                      (cond [z? fail]
                            [(and (num? n₀) (num? n₁)) (return (/ n₀ n₁))]
                            [else (return `(/ ,n₀ ,n₁))]))] ...))
 (define (zero? v)
   (do ϕ ← get-path-cond
       (match v
         [(? num? n)             (return (= 0 n))]
         [v #:when (∈ v ϕ)       (return #t)]
         [v #:when (∈ `(¬ ,v) ϕ) (return #f)]
         [v (mplus (do (refine v)       (return #t))
                   (do (refine `(¬ ,v)) (return #f)))])))
```

Figure 11: Symbolic Execution Variant

```
 > (rec id (λ (n) (if0 n 0 (+ 1 (id (- n 1)))))
     (id 3))
 with-limit: out of time
```

To ensure termination for all programs, we assume all references to primitive operations are η-expanded, so that store-allocations also take place at primitive applications, ensuring widening at repeated bindings. In fact, all programs terminate when using `precise-δ@`, `store-crush@` and η-expanded primitives, which means we have a achieved a computable and uniquely precise abstract interpreter.

Here we see one of the strengths of the extensible, definitional approach to abstract interpreters. The combination of added precision and widening is encoded quite naturally. In contrast, it's hard to imagine how such a combination could be formulated as, say, a constraint-based flow analysis.

8 SYMBOLIC EXECUTION

In this section, we carry out another—this time more involved—example that shows how to instantiate our definitional abstract interpreter to obtain a symbolic execution engine that performs sound program verification. This serves to demonstrate the range of the approach, capturing forms of analysis typically considered fairly dissimilar.

```
                                                              δ^-symbolic@
(define (δ o n₀ n₁)
  (match* (o n₀ n₁)
    [('/ n₀ n₁) (do z? ← (zero? n₁)
                    (cond [z? fail]
                          [(member 'N (list n₀ n₁)) (return 'N)]
                          ...))]
    ...))
(define (zero? v)
  (do ϕ ← get-path-cond
      (match v ['N (mplus (return #t) (return #f))] ...)))
```

Figure 12: Symbolic Execution with Abstract Numbers

First, we describe the monad stack and metafunctions that implement a symbolic executor (C. 1976), then we show how abstractions discussed in previous sections can be applied to enforce termination, turning a traditional symbolic execution into a path-sensitive verification engine.

To support symbolic execution, the syntax of the language is extended to include symbolic numbers:

$e \in exp$ ::= ... | (sym x) [*symbolic number*]
$\varepsilon \in pexp$ ::= $e \mid \neg e$ [*path expression*]
$\phi \in pcon$ ::= $P(pexp)$ [*path condition*]

Figure 12 shows the units needed to turn the existing interpreter into a symbolic executor. Primitives such as `'/` now also take as input and return symbolic values. As standard, symbolic execution employs a path-condition accumulating assumptions made at each branch, allowing the elimination of provably infeasible paths and construction of test cases. We represent the path-condition ϕ as a set of symbolic values or their negations. If `e` is in ϕ, `e` is assumed to evaluate to `0`; if `¬ e` is in ϕ, `e` is assumed to evaluate to non-`0`. This set is another state component provided by `StateT` in the monad transformer stack. Monadic operations `get-path-cond` and `refine` reference and update the path-condition. The metafunction `zero?` works similarly to the concrete counterpart, but also uses the path-condition to prove that some symbolic numbers are definitely `0` or non-`0`. In case of uncertainty, `zero?` returns both answers instead of refining the path-condition with the assumption made.

In the following example, the symbolic executor recognizes that result `3` and division-by-0 error are not feasible:

```
> (if0 'x (if0 'x 2 3) (quotient 5 'x))
(set
 (cons '(quotient 5 x) (set '(¬ x)))
 (cons 2 (set 'x)))
```

A scaled up symbolic executor could implement `zero?` by calling out to an SMT solver for more interesting reasoning about arithmetic, or extend the language with symbolic functions and blame semantics for sound higher-order symbolic execution, essentially recreating a pushdown variant of Nguyễn et al. (C. and David 2015; C. et al. 2014; Sam and David 2012).

Traditional symbolic executors mainly aim to find bugs and do not provide a termination guarantee. However, when we apply to this symbolic executor the finite abstractions presented in previous sections, namely base value

```
                                                              monad-pdcfa-gc@
(define-monad (ReaderT (ReaderT (FailT (StateT (NondetT (ReaderT (StateT+ ID))))))))
                                                               mrun-pdcfa-gc@
(define (mrun m)
  (run-StateT+ ∅ (run-ReaderT ∅   ; out-$₀, in-$₀
  (run-StateT  ∅ (run-ReaderT ∅   ; σ₀, ρ₀
  (run-ReaderT (set) m))))))      ; ψ₀
```

Figure 13: Monad Instance with Root Address Set

widening and finite allocation (section 3.2), and caching and fixing (section 4), we turn the symbolic execution into a sound, path-sensitive program verification engine.

There is one wrinkle, which is that operations on symbolic values introduce a new source of unboundness in the state-space, because the space of symbolic values is not finite. A simple strategy to ensure termination is to widen a symbolic value to the abstract number 'N when it shares an address with a different number, similarly to the precision-preserving abstraction from section 7. Figure 12 shows extension to δ and `zero?` in the presence of `'N`. The different treatments of `'N` and symbolic values clarifies that abstract values are not symbolic values: the former stands for a set of multiple values, whereas the latter stands for an single unknown value. Tests on abstract number `'N` do not strengthen the path-condition; it is unsound to accumulate any assumption about `'N`.

9 GARBAGE COLLECTION

As a denouement to our series of examples, we show how to incorporate garbage collection into our definitional abstract interpreter.

This example, like store-widening, is the re-creation of a well-known technique: abstract garbage collection (Matthew and Olin 2006b) mimics the process of reclaiming unreachable heap addresses as done in garbage-collecting concrete interpreters. While garbage collection in the concrete can largely be considered an implementation detail that doesn't effect the results of computation (modulo pragmatic issues of memory consumption), in the abstract semantics, garbage collection can have a significant positive effect on the precision of analysis results. This is because store locations mediate joins, and therefore points of imprecision, in the abstract semantics. If an address can be cleared-out and recycled, this represents the avoidance of a join that would be encountered in a non-garbage-collecting abstract interpreter.

In the finite-state-machine model, abstract garbage collection is fairly straightforward and closely follows concrete formulations (David and Matthew 2010; Matthew and Olin 2006b). However, incorporating both pushdown control flow and abstract garbage collection has proved rather involved and required new techniques (Christopher et al. 2012; Ian et al. 2014).

The key difficulty for pushdown machine models, which essentially use abstract models that are pushdown automata, is that the usual approach to garbage collection is to crawl the call stack to compute the root set of reachable addresses (Greg et al. 1995). Traversing the stack, however, is not something that can be expressed by a pushdown automata.

This difficulty is somewhat exacerbated by the definitional interpreter approach in that there isn't even a stack to traverse! Nevertheless, as we demonstrate, this challenge can be overcome to obtain a pushdown, garbage-collecting abstract interpreter.

Doing so shows that the definitional abstract interpreter approach also scales to handle so-called *introspective* pushdown analysis that require some level of introspection on the stack (Christopher et al. 2012; Ian et al. 2014).

Solving the abstract garbage collection problem for a definitional abstract interpreter boils down to answer the following question: how can we track root addresses that are live on the call stack when the call stack is implicitly defined by the metalanguage? The answer is fairly simple: we extend the monad with a set of root addresses. When evaluating compound expressions, we calculate the appropriate root sets for the context. In essence, we render explicit only the addresses of the calling context, while still relying on the metalanguage to implicitly take care of the rest as before.

Figure 13 defines the appropriate monad instance. All that has changed is there is an added reader component, which will be used to model the context's current root set.

The use of this added component necessitates a change to the caching and fixed-point calculation, namely we must include the root sets as part of the configuration. Compared with the `ev-cache@` component of section 4, we make a simple adjustment to the first few lines to cache the root set along with the rest of the configuration:

```
(define (((ev-cache ev₀) ev) e)
  (do ρ    ← ask-env   σ ← get-store   ψ ← ask-roots
      ς    ≔ (list e ρ σ ψ)
      ...))
```

Similarly, for `fix-cache@`:

```
(define ((fix-cache eval) e)
  (do ρ ← ask-env   σ ← get-store   ψ ← ask-roots
      ς ≔ (list e ρ σ ψ)
      ...))
```

We can now write a `ev-collect@` component that performs the garbage collection: it asks for the current roots in the context, evaluates an expression to a value, then updates the store after garbage collecting all addresses not reachable from the root set of the context and the roots in the produced value:

```
(define (((ev-collect ev0) ev) e)
  (do ψ ← ask-roots
      v ← ((ev0 ev) e)
      (update-store (gc (set-union ψ (roots-v v))))
      (return v)))
```

Here, `gc` and `roots-v` are (omitted) helper functions that perform garbage collection and calculate the set of root addresses in a value, respectively.

All that remains is to define a component that propagates root sets appropriately from compound expressions to their constituents. Figure 14 gives the `ev-roots@` component, which does exactly this.

Finally, the pieces are stitched together with the following to obtain a pushdown, garbage-collecting definitional abstract interpreter:

```
(define (eval e)
  (mrun ((fix-cache (fix (ev-cache (ev-collect (ev-roots ev))))) e)))
```

10 RELATED WORK

This work draws upon and re-presents many ideas from the literature on abstract interpretation for higher-order languages (Jan 2012). In particular, it closely follows the abstracting abstract machines (David and Matthew 2010, 2012) approach to deriving abstract interpreters from a small-step machine. The key difference here is that we operate in the setting of a monadic definitional interpreter instead of an abstract machine. In moving to this new setting we developed a novel caching mechanism and fixed-point algorithm, but otherwise followed the same

```
                                                                      ev-roots@
(define (((ev-roots ev₀) ev) e)
  (match e
    [(if0 e₀ e₁ e₂) (do ψ  ← ask-roots
                        ρ  ← ask-env
                        ψ′ ≔ (set-union ψ (roots e₁ ρ) (roots e₂ ρ))
                        v  ← (local-roots ψ′ (ev e₀))
                        b  ← (truish? v)
                        (ev (if b e₁ e₂)))]
    [(op2 o e₀ e₁)  (do ψ  ← ask-roots
                        ρ  ← ask-env
                        v₀ ← (local-roots (set-union ψ (roots e₁ ρ)) (ev e₀))
                        v₁ ← (local-roots (set-union ψ (roots-v v₀)) (ev e₁))
                        (δ o v₀ v₁))]
    [(app e₀ e₁)    (do ρ  ← ask-env
                        ψ  ← ask-roots
                        v₀ ← (local-roots (set-union ψ (roots e₁ ρ)) (ev e₀))
                        v₁ ← (local-roots (set-union ψ (roots-v v₀)) (ev e₁))
                        (cons (lam x e₂) ρ′) ≔ v₀
                        a  ← (alloc x)
                        (ext a v₁)
                        (local-env (ρ′ x a) (ev e₂)))]
    [_ ((ev₀ ev) e)]))
```

Figure 14: Address Collection and Propagation

recipe. Remarkably, in the setting of definitional interpreters, the pushdown property for the analysis is simply inherited from the meta-language rather than requiring explicit instrumentation to the abstract interpreter.

Compositionally defined abstract interpretation functions for higher-order languages were first explored by D. and Flemming (1995), which introduces the technique of interpreting a higher-order object language directly as terms in a meta-language to perform abstract interpretation. While their work lays the foundations for this idea, it does not consider abstractions for fixed-points in the domain, so although their abstract interpreters are sound, they are not in general computable. They propose a naïve solution of truncating the interpretation of syntactic fixed-points to some finite depth, but this solution isn't general and doesn't account for non-syntactic occurrences of bottom in the concrete domain (*e.g. via* Y combinators). Our work develops such an abstraction for concrete denotational fixed-points using a fixed-point caching algorithm, resulting in general, computable abstractions for arbitrary definitional interpreters.

The use of monads and monad transformers to make extensible (concrete) interpreters is a well-known idea (Eugenio 1989; L. 1994; Sheng et al. 1995), which we have extended to work for compositional abstract interpreters. The use of monads and monad transformers in machine based-formulations of abstract interpreters has previously been explored by Ilya et al. (2013) and David et al. (2015), respectively, and inspired our own adoption of these ideas. Darais has also shown that certain monad transformers are also *Galois transformers*, i.e. they compose to form monads that transport Galois connections. This idea may pave a path forward for obtaining both compositional code *and proofs* for abstract interpreters in the style presented here.

The caching mechanism used to ensure termination in our abstract interpreter is similar to that used by J.∼Ian and David (2014). They use a local- and meta-memoization table in a machine-based interpreter to ensure termination for a pushdown abstract interpreter. This mechanism is in turn reminiscent of Glück's use of memoization in an interpreter for two-way non-deterministic pushdown automata (Robert 2013).

Caching recursive, non-deterministic functions is a well-studied problem in the functional logic programming community under the rubric of "tabling" (Hisao and Taisuke 1986; N and Lars 1993; Terrance and S 2012; Weidong and S 1996), and has been usefully applied to program verification and analysis (Gerda and F 1998; Steven et al. 1996). Unlike these systems, our approach uses a shallow embedding of cached non-determinism that can be applied in general-purpose functional languages. % Monad transformers that enable shallow embedding of cached non-determinism are of continued interest since Hinze's *Deriving Backtracking Monad Transformers* (Oleg et al. 2005; Ralf 2000; Sebastian et al. 2011), and recent work (Alexander et al. 2016; der and Oleg 2014) points to potential optimizations and specializations that can be applied to our relatively naive iteration strategy.

Vardoulakis, who was the first to develop the idea of a pushdown abstraction for higher-order flow analysis (Dimitrios and Olin 2011), formalized CFA2 using a CPS model, which is similar in spirit to a machine-based model. However, in his dissertation (Vardoulakis 2012) he sketches an alternative presentation dubbed "Big CFA2" which is a big-step operational semantics for doing pushdown analysis quite similar in spirit to the approach presented here. One key difference is that Big CFA2 fixes a particular coarse abstraction of base values and closures—for example, both branches of a conditional are always evaluated. Consequently, it only uses a single iteration of the abstract evaluation function, and avoids the need for the cache-based fixed-point of section 4. We believe Big CFA2 as stated is sound, however if the underlying abstractions were tightened, it may then require a more involved fixed-point finding algorithm like the one we developed.

Our formulation of a pushdown abstract interpreter computes an abstraction similar to the many existing variants of pushdown flow analysis (Christopher et al. 2012; Christopher et al. 2010; David and Matthew 2012; Dimitrios and Olin 2011; Ian et al. 2014; J.∼Ian and David 2014; Thomas et al. 2016; Vardoulakis 2012).

The mixing of symbolic execution and abstract interpretation is similar in spirit to the *logic flow analysis* of Might (Matthew 2007b), albeit in a pushdown setting and with a stronger notion of negation; generally, our presentation resembles traditional formulations of symbolic execution more closely (C. 1976). Our approach to symbolic execution only handles the first-order case of symbolic values, as is common. However, Nguyễn's work on higher-order symbolic execution (C. and David 2015) demonstrates how to scale to behavioral symbolic values. In principle, it should be possible to handle this case in our approach by adapting Nguyễn's method to a formulation in a compositional evaluator, but this remains to be carried out.

Now that we have abstract interpreters formulated with a basis in abstract machines and with a basis in monadic interpreters, an obvious question is can we obtain a correspondence between them similar to the functional correspondence between their concrete counterparts (S. et al. 2005). An interesting direction for future work is to try to apply the usual tools of defunctionalization, CPS, and refocusing to see if we can interderive these abstract semantic artifacts.

11 CONCLUSIONS

We have shown that the AAM methodology can be adapted to definitional interpreters written in monadic style. Doing so captures a wide variety of semantics, such as the usual concrete semantics, collecting semantics, and various abstract interpretations. Beyond recreating existing techniques from the literature such as store-widening and abstract garbage collection, we can also design novel abstractions and capture disparate forms of program analysis such as symbolic execution. Further, our approach enables the novel combination of these techniques.

To our surprise, the definitional abstract interpreter we obtained implements a form of pushdown control flow abstraction in which calls and returns are always properly matched in the abstract semantics. True to the definitional style of Reynolds, the evaluator involves no explicit mechanics to achieve this property; it is simply inherited from the metalanguage.

We believe this formulation of abstract interpretation offers a promising new foundation towards re-usable components for the static analysis and verification of higher-order programs. Moreover, we believe the definitional abstract interpreter approach to be a fruitful new perspective on an old topic. We are left wondering: what else can be profitably inherited from the metalanguage of an abstract interpreter?

BIBLIOGRAPHY

Vandenbroucke, Alexander, Schrijvers, Tom, and Piessens, Frank. Fixing non-determinism. In *Proc. IFL 2015: Symposium on the implementation and application of functional programming languages Proceedings*, 2016.

King, James C. Symbolic Execution and Program Testing. *Commun. ACM*, 1976.

Nguyễn, Phúc C. and Van Horn, David. Relatively Complete Counterexamples for Higher-order Programs. In *Proc. Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

Nguyễn, Phúc C., Tobin-Hochstadt, Sam, and Van Horn, David. Soft Contract Verification. In *Proc. Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, 2014.

Reynolds, John C. Definitional interpreters for higher-order programming languages. In *Proc. ACM '72: Proceedings of the ACM Annual Conference*, 1972.

Earl, Christopher, Sergey, Ilya, Might, Matthew, and Van Horn, David. Introspective pushdown analysis of higher-order programs. In *Proc. Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, 2012.

Earl, Christopher, Might, Matthew, and Van Horn, David. Pushdown control-flow analysis of higher-order programs. In *Proc. Workshop on Scheme and Functional Programming*, 2010.

Jones, Neil D. and Nielson, Flemming. Abstract Interpretation: A Semantics-based Tool for Program Analysis. In *Proc. Handbook of Logic in Computer Science (Vol. 4)*, 1995.

Darais, David, Might, Matthew, and Van Horn, David. Galois Transformers and Modular Abstract Interpreters: Reusable Metatheory for Program Analysis. In *Proc. Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

Van Horn, David and Might, Matthew. Abstracting Abstract Machines. In *Proc. Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, 2010.

Van Horn, David and Might, Matthew. Systematic abstraction of abstract machines. *Journal of Functional Programming*, 2012.

Ploeg, Atze van der and Kiselyov, Oleg. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In *Proc. Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, 2014.

Vardoulakis, Dimitrios and Shivers, Olin. CFA2: a Context-FreeApproach to Control-FlowAnalysis. *Logical Methods in Computer Science*, 2011.

Christopher Earl. Introspective Pushdown Analysis and Nebo. University of Utah, 2014.

Moggi, Eugenio. An Abstract View of Programming Languages. Edinburgh University, 1989.

Janssens, Gerda and Sagonas, Konstantinos F. On the Use of Tabling for Abstract Interpretation: An Experiment with Abstract Equation Systems. In *Proc. TAPD*, 1998.

Morrisett, Greg, Felleisen, Matthias, and Harper, Robert. Abstract Models of Memory Management. In *Proc. Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, 1995.

Tamaki, Hisao and Sato, Taisuke. OLD resolution with tabulation. In *Proc. International Conference on Logic Programming*, 1986.

Johnson, J. Ian, Sergey, Ilya, Earl, Christopher, Might, Matthew, and Van Horn, David. Pushdown flow analysis with abstract garbage collection. *Journal of Functional Programming*, 2014.

Sergey, Ilya, Devriese, Dominique, Might, Matthew, Midtgaard, Jan, Darais, David, Clarke, Dave, and Piessens, Frank. Monadic Abstract Interpreters. In *Proc. Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

Johnson, J.∼Ian and Van Horn, David. Abstracting Abstract Control. In *Proc. Proceedings of the 10th ACM Symposium on Dynamic Languages*, 2014.

Midtgaard, Jan. Control-flow Analysis of Functional Programs. *ACM Comput. Surv.*, 2012.

Midtgaard, Jan and Jensen, Thomas P. Control-flow Analysis of Function Calls and Returns by Abstract Interpretation. In *Proc. ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, 2009.

Midtgaard, Jan and Jensen, Thomas. A Calculational Approach to Control-FlowAnalysis by Abstract Interpretation. In *Proc. Static Analysis Symposium*, 2008.

Jaskelioff, Mauro Javier. Lifting of operations in modular monadic semantics. University of Nottingham, 2009.

Wright, Andrew K. and Jagannathan, Suresh. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Trans. Program. Lang. Syst.*, 1998.

Steele,Jr., Guy L. Building Interpreters by Composing Monads. In *Proc. Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1994.

Flatt, Matthew and Felleisen, Matthias. Units: Cool Modules for HOT Languages. In *Proc. Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, 1998.

Flatt, Matthew and PLT. Reference: Racket. PLT Inc., 2010.

Might, Matthew. Environment analysis of higher-order languages. Georgia Institute of Technology, 2007a.

Might, Matthew. Logic-flow Analysis of Higher-order Programs. In *Proc. Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007b.

Might, Matthew and Van Horn, David. A Family of Abstract Interpretations for Static Analysis of Concurrent Higher-OrderPrograms. In *Proc. Static Analysis*, 2011.

Might, Matthew and Shivers, Olin. Environment analysis via \(\Delta\)-CFA. In *Proc. POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006a.

Might, Matthew and Shivers, Olin. Improving Flow Analyses via \(\Gamma\)CFA: Abstract Garbage Collection and Counting. In *Proc. ICFP'06*, 2006b.

Bol, Roland N and Degerstedt, Lars. Tabulated Resolution for Well Founded Semantics. In *Proc. ILPS*, 1993.

Kiselyov, Oleg. Typed tagless final interpreters. In *Proc. Generic and Indexed Programming*, 2012.

Kiselyov, Oleg, Shan, Chung-chieh, Friedman, Daniel P., and Sabry, Amr. Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl). In *Proc. Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, 2005.

Shivers, Olin. Control-flow analysis of higher-order languages. Carnegie Mellon University, 1991.

Hinze, Ralf. Deriving Backtracking Monad Transformers. In *Proc. Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, 2000.

Glück, Robert. Simulation of Two-Way Pushdown Automata Revisited. In *Proc. Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013*, 2013.

Ager, Mads S., Danvy, Olivier, and Midtgaard, Jan. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 2005.

Tobin-Hochstadt, Sam and Van Horn, David. Higher-order symbolic execution via contracts. In *Proc. Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2012.

Tobin-Hochstadt, Sam, St-Amour, Vincent, Culpepper, Ryan, Flatt, Matthew, and Felleisen, Matthias. Languages As Libraries. In *Proc. Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

Fischer, Sebastian, Kiselyov, Oleg, and Shan, Chung-chieh. Purely functional lazy nondeterministic programming. *Journal of Functional Programming* 21(4-5), pp. 413–465, 2011.

Liang, Sheng, Hudak, Paul, and Jones, Mark. Monad Transformers and Modular Interpreters. In *Proc. Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995.

Dawson, Steven, Ramakrishnan, C. R., and Warren, David S. Practical Program Analysis Using General Purpose Logic Programming Systems—a Case Study. In *Proc. Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, 1996. http://doi.acm.org/10.1145/231379.231399

Jagannathan, Suresh, Thiemann, Peter, Weeks, Stephen, and Wright, Andrew. Single and loving it: must-alias analysis for higher-order languages. In *Proc. POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.

Jagannathan, Suresh and Weeks, Stephen. A unified treatment of flow analysis in higher-order languages. In *Proc. POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995.

Swift, Terrance and Warren, David S. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming* 12(1-2), pp. 157–187, 2012.

Gilray, Thomas, Lyde, Steven, Adams, Michael D., Might, Matthew, and Van Horn, David. Pushdown Control-flow Analysis for Free. In *Proc. Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

Dimitrios Vardoulakis. CFA2: Pushdown Flow Analysis for Higher-Order Languages. Northeastern University, 2012.

Chen, Weidong and Warren, David S. Tabled evaluation with delaying for general logic programs. *Journal of the ACM (JACM)* 43(1), pp. 20–74, 1996.