Abstracting Definitional Interpreters

David Darais
University of Maryland
darais@cs.umd.edu

Nicholas Labich
University of Maryland
labichn@cs.umd.edu

Phúc C. Nguyễn
University of Maryland
pcn@cs.umd.edu

David Van Horn
University of Maryland
dvanhorn@cs.umd.edu

Abstract

In this paper we show that definitional interpreters for programming languages can express not only the usual notion of interpretation, but also a wide variety of collecting semantics, abstract interpretations, symbolic executions and their intermixings. To achieve this we reconstruct a standard definitional interpreter using two new ingredients: monadic operations and open recursion. The resulting definitional *abstract* interpreter is extensible and we recover various abstract semantics through its instantiations.

In addition to recovering well-known abstractions, we show that the resulting abstract interpreter is perfectly precise in modeling call and return flows, *i.e.* it implements Pushdown Control Flow Analysis (PDCFA). True to the definitional style of Reynolds, the abstract interpreter contains no explicit mechanics to achieve this property; it is simply inherited from the defining metalanguage.

In addition to demonstrating our technique, we formalize a systematic methodology for deriving definitional abstract interpreters, prove it sound, and make precise its relationship to PDCFA.

1. Introduction

In his landmark paper, *Definitional Interpreters for Higher-order Programming Languages* [30], Reynolds observed that when a programming language is defined by way of an interpreter, it is possible to inherit semantic characteristics of the defining metalanguage. For example, it is possible to define a definitional interpreter which is call-by-value when the metalanguage is call-by-value, and call-by-name when the metalanguage is call-by-name.

We expand on Reynolds's observation in the setting of abstract interpreters and discover the following:

- Definitional interpreters, written in monadic style, can simultaneously define a language's semantics as well as safe approximations of those semantics, *i.e.* abstract interpreters.
- These definitional *abstract* interpreters can inherit characteristics of the defining language. In particular, precise call-and-return matching can be inherited, yielding a pushdown analysis [6, 41].

Furthermore, we contribute:

- A systematic methodology for designing abstract interpreters via definitional interpreters; and
- A soundness framework establishing the correctness of abstract interpreters defined in definitional style.

A Compositional Approach to Program Analysis There are many approaches to designing program analyzers for programming languages. For *higher-order* programming languages, two popular approaches are to use abstract machines [38] and constraint systems [28]. Other foundations exist for designing program analyzers, but no approach to-date utilizes big-step operational semantics or definitional interpreters for program analysis of higher-order languages. This is unfortunate because big-step semantics and definitional interpreters are more compositional and high-level than state machines or constraint systems. Big-step and denotational approaches exist for first-order programming languages; the gap is in extending these ideas to the higher-order setting.

We bridge this gap, providing the first framework for program analysis of higher-order languages which builds upon big-step semantics and their corresponding definitional interpreters, which are compositional by nature.

Our key insights are to design definitional interpreters in monadic, open-recursive style (§ 3), and to design a novel fixpoint algorithm tailored specifically to the setting of higher-order definitional interpreters (§ 4). The extensible nature of the interpreter allows us to recover a wide-range of analyses through its instantiation, including widening techniques (§ 6), precision preserving abstractions (§ 7), and symbolic execution for program verification (§ 8). Our implementation is freely available through a suite of embedded languages in Racket (§ 9). Finally, we prove the approach sound w.r.t. a derived big-step collecting and abstract semantics (§ 10), where the key insight in the formalism is to model not only standard big-step *evaluation* relations, but also big-step *reachability* relations.

Pushdown Precision in Program Analysis A common problem with traditional approaches to control flow analysis is the inability to properly match a function call with its return in the abstract semantics. This leads to infeasible program (abstract) executions in which a call is made from one point in the program, but control returns to another. The PDCFA analysis of Earl *et al* [6] and CFA2 analysis of Vardoulakis and Shivers [42] were the first approaches to overcome this limitation in the higher-order setting. In essence, these analyses replaces the traditional finite automata abstractions of programs with pushdown automata, an approach pioneered by Reps *et al* [29] in the first-order setting.

We realize a new technique for defining abstract interpreters with pushdown precision, meaning the analysis precisely matches function calls to returns. In the setting of definitional interpreters, this property is inherited from the defining metalanguage and requires no instrumentation to the analysis *at all* (§ 5).

A technical difference between small-step and big-step approaches to semantics are that small-step methods must model the execution context of evaluation, whether through evaluation contexts [9] or stack frames [8]. In big-step methods, there is no model

$$
\begin{array}{rrll}
e \in & exp ::= & \texttt{(vbl } x\texttt{)} & [\textit{variable}] \\
& | & \texttt{(num } n\texttt{)} & [\textit{number}] \\
& | & \texttt{(lam } x\ \texttt{)} & [\textit{lambda}] \\
& | & \texttt{(ifz } e\ e\ e\texttt{)} & [\textit{conditional}] \\
& | & \texttt{(op2 } b\ e\ e\texttt{)} & [\textit{binary op}] \\
& | & \texttt{(app } e\ e\texttt{)} & [\textit{application}] \\
& | & \texttt{(rec } x\ e\ e\texttt{)} & [\textit{letrec}] \\
x \in & var ::= & \{\texttt{x}, \texttt{y}, \ldots\} & [\textit{variable names}] \\
b \in & binop ::= & \{\texttt{+}, \texttt{-}\ , \ldots\} & [\textit{binary prim}]
\end{array}
$$

Figure 1. Programming Language Syntax

for the context; it is implicit in the definition of evaluation rules, or in the case of definitional interpreters, implicit in the call-and-return semantics of the defining programming language. Because a defining programming language is precise in its call/return behavior, so is a definitional abstract interpreter embedded within it.

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 [38] 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 whether a similar approach is possible with a higher-level formulation of the semantics, such as a compositional evaluation function, or so-called big-step semantics.

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 [17, 21, 24], 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 in ways that were not available to the small-step machine world.

ev@

```
(define ((ev ev) e)
  (match e
    [(num n)        (return n)]
    [(vbl x)        (do ρ ← ask-env
                        (find (ρ x)))]
    [(ifz 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 e₀ e₁)  (do ρ  ← ask-env
                        a  ← (alloc f)
                        ρ′ := (ρ f a)
                        (ext a (cons e₀ ρ′))
                        (local-env ρ′ (ev e₁)))]
    [(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

3. A Definitional Interpreter

We begin by constructing a definitional interpreter for a small but representative higher-order, functional language. As our defining language, we use an applicative subset of Racket, a dialect of Scheme.[1] 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.
- 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 `(ρ x)` denotes $\rho(x)$ while `(ρ x a)` denotes $\rho[x \mapsto a]$. 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

[1] This choice is largely immaterial: any functional language would do.

monad@

```
                    env     errors   store
(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

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, noted in italics, which must 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.[2]

3.1 Instantiating the Interpreter

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 [23].

We also define mrun 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]))
```

[2] We use Racket *units* [11] to model components in our implementation.

trace-monad@

```
(define-monad
      env     errors   store    traces
   (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

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. Finally 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 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, which is straightforward to define:

```
(define ((fix f) x) ((f (fix f)) x))
```

and the main entry-point for the interpreter:

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

By taking advantage of Racket's languages-as-libraries features [36], we construct REPLs for interacting with this interpreter. Here are a few examples, which make use of a concrete syntax for more succinctly writing expressions. The identity function evaluates to a closure over the empty environment paired with the empty store:

```
> (λ (x) x)
'(((λ (x) x) . ()) . ())
```

Here's an example showing 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 . ())
```

And divide-by-zero errors result in failures:

```
> (/ 5 (- 3 3))
'(failure . ())
```

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:

```
> (/ 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.

3.2 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

```
(define-monad                                                  dead-monad@
      env     store    dead    errors
   (ReaderT (StateT (StateT (FailT ID)))))

(define (((ev-dead ev₀) ev) e)                                 ev-dead@
  (do θ ← get-dead
      (put-dead (set-remove θ e))
      ((ev₀ ev) e)))

(define ((eval-dead eval) e₀)                                  eval-dead@
  (do (put-dead (subexps e₀))
      (eval e₀)))
```

Figure 5. Dead Code Collecting Semantics

```
(define-monad                                                  monad^@
      env     errors   store    mplus
   (ReaderT (FailT (StateT (NondetT ID)))))

(define (δ o n₀ n₁)                                            δ^@
  (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

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-trace` function is a wrapper around an underlying $\texttt{ev}_0$ 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 (not showing ρ or σ in results):

```
> (* (+ 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 $\texttt{ev}_0$ 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 (/ 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.

3.3 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 `'/`. 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 representation 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 . ()))
> (/ 5 (+ 1 2))
'((failure . ()) (N . ()))
> (if0 (+ 1 0) 3 4)
'((3 . ()) (4 . ()))
```

If we link `δ^@` with the *tracing* monad stack plus non-determinism:

```
      env     errors   store     traces          mplus
 (ReaderT (FailT (StateT (WriterT List (NondetT ID)))))
```

we get an evaluator that produces sets of traces (again not showing ρ or σ in the results):

```
> (if0 (+ 1 0) 3 4)
(set '((3 . ()) (if0 (+ 1 0) 3 4) (+ 1 0) 0 3)
     '((4 . ()) (if0 (+ 1 0) 3 4) (+ 1 0) 0 4))
```

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.

3.4 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

```
(define (alloc x) (return x))                                  alloc^@

(define (find a)                                               store-nd@
  (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

finitize closures by finitizing 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 Fixpoints

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))
timeout
```

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

```
                                  env   errors                   monad-cache@
(define-monad (ReaderT (FailT
     store    mplus     in-$     out-$
   (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

fixed-point of $\$^{+}$ of an evaluator which transforms an oracle $\$^{in}$ and outputs a more defined oracle $\$^{out}$ is then a sound approximation of the program, because it over-approximate all finite-number of unrollings of the unfixed evaluator. We formalize this co-inductive process in Section 10 and prove it sound; in this section we instead focus on the intuition and implementation for the algorithm.

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 $(\$^{in}\ \varsigma)$, which is the "oracle" result. Also, whenever we finish computing the result $v \times \sigma$ of evaluating a configuration ς, we place an entry in the output cache mapping ς to $v \times \sigma'$. 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 fixpoint 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.

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 Fixpoints in the Cache

With these peices in place we construct a complete interpreter:

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

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)))
'(0 2 3)
```

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!

The abstract machine formulation of the semantics models 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. Like closures, the AAM trick is to thread them through the store and then finitize the store. But in the definitional interpreter approach, the stack is implicit and 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 a dissertation [6, 7, 13, 18, 19, 39, 41, 42]. Yet when formulated in the definitional interpreter style, the pushdown property requires no mechanics and is simply inherited from the meta-language.

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 (zero? 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

Reynolds, in his celebrated paper *Definitional Interpreters for Higher-order Programming Languages* [30], 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

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 "widenened" store, 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 Darais *et al* [4]. Whereas before we had `monad-cache@` we instead swap the order of `StateT` for the store and `NondetT`:

```
          env      errors    mplus     store
(ReaderT (FailT (NondetT (StateT+
                in−$      out−$
  (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.

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

```
(define-monad                                          symbolic-monad@
      env    errors   store    path    mplus
   (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

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.

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, "straight-line" arithmetic operations are computed with full precision:

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

Even linear binding and arithmetic preserves precision:

```
> ((λ (x) (* x x)) 5)
'(25)
```

It's 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.

```
> (rec id (λ (n) (if0 n 0 (+ 1 (id (- n 1)))))
    (id 3))
timeout
```

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

```
                                                       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

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. Path Sensitivity

As a final exercise in applying our definitional abstract interpretation framework, we develop a symbolic execution engine, and use it to perform sound program verification. First we describe the monad stack and metafunctions that implement a symbolic executor [20], then show how abstractions discussed in previous sections can be applied to enforce termination, turning a traditional symbolic execution into a path-sensitive verification engine.

8.1 Symbolic Execution

To support symbolic execution, first we extend the syntax of the language to support symbolic numbers:

$$\begin{array}{lll} e \in \ exp ::= \ \ldots \mid (\texttt{sym}\ x) & [\textit{symbolic number}] \\ \epsilon \in pexp ::= \ e \mid \neg e & [\textit{path expression}] \\ \phi \in \ pcon \coloneqq \ \wp(pexp) & [\textit{path condition}] \end{array}$$

Figure 8 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 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) (/ 5 'x))
(set (cons '(/ 5 x) (set '(¬ x)))
     (cons 2 (set 'x)))
```

A scaled up symbolic executor could implement *`zero?`* by calling out to an SMT solver for interesting arithmetics, or extend the language with symbolic functions and blame semantics for sound higher-order symbolic execution [27, 35].

[*Concrete Evaluation*] $\boxed{\rho, \tau \vdash e, \sigma \Downarrow v, \sigma'}$

$$\text{(Lit)}\ \frac{}{\rho, \tau \vdash n, \sigma \Downarrow n, \sigma} \qquad \text{(Var)}\ \frac{}{\rho, \tau \vdash x, \sigma \Downarrow \sigma(\rho(x)), \sigma} \qquad \text{(Lam)}\ \frac{}{\rho, \tau \vdash \lambda x.e, \sigma \Downarrow \langle \lambda x.e, \rho \rangle, \sigma}$$

$$\text{(Bin)}\ \frac{\rho, \tau \vdash e_1, \sigma \Downarrow v_1, \sigma_1 \qquad \rho, \tau \vdash e_2, \sigma_1 \Downarrow v_2, \sigma_2}{\rho, \tau \vdash b(e_1, e_2), \sigma \Downarrow [\![b]\!](v_1, v_2), \sigma_2}$$

$$\text{(App)}\ \frac{\rho, \tau \vdash e_1, \sigma \Downarrow v_1, \sigma_1 \qquad \rho, \tau \vdash e_2, \sigma_1 \Downarrow v_2, \sigma_2 \qquad \rho'[x \mapsto \ell], \tau' \vdash e', \sigma_2[\ell \mapsto v_2] \Downarrow v', \sigma_3}{\rho, \tau \vdash e_1(e_2), \sigma \Downarrow v', \sigma_3} \begin{array}{r} \langle \lambda x.e', \rho' \rangle = v_1 \\ \ell = \langle x, \tau' \rangle \\ \tau' \quad \textit{fresh} \end{array}$$

$$\text{(IfT)}\ \frac{\rho, \tau \vdash e_1, \sigma \Downarrow n, \sigma_1 \qquad \rho, \tau \vdash e_2, \sigma_1 \Downarrow v, \sigma_2}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \sigma \Downarrow v, \sigma_2}\ n = 0 \qquad \text{(IfF)}\ \frac{\rho, \tau \vdash e_1, \sigma \Downarrow n, \sigma_1 \qquad \rho, \tau \vdash e_3, \sigma_1 \Downarrow v, \sigma_2}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \sigma \Downarrow v, \sigma_2}\ n \neq 0$$

[*Concrete Reachability*] $\boxed{\rho, \tau \vdash e, \sigma \Uparrow \varsigma}$

$$\text{(Refl)}\ \frac{}{\rho, \tau \vdash e, \sigma \Uparrow \langle e, \rho, \sigma, \tau \rangle} \qquad \text{(RBin1)}\ \frac{\rho, \tau \vdash e_1, \sigma \Uparrow \varsigma}{\rho, \tau \vdash b(e_1, e_2), \sigma \Uparrow \varsigma} \qquad \text{(RBin2)}\ \frac{\rho, \tau \vdash e_1, \sigma \Downarrow v_1, \sigma_1 \qquad \rho, \tau \vdash e_2, \sigma_1 \Uparrow \varsigma}{\rho, \tau \vdash b(e_1, e_2), \sigma \Uparrow \varsigma}$$

$$\text{(RApp1)}\ \frac{\rho, \tau \vdash e_1, \sigma \Uparrow \varsigma}{\rho, \tau \vdash e_1(e_2), \sigma \Uparrow \varsigma} \qquad \text{(RApp2)}\ \frac{\rho, \tau \vdash e_1, \sigma \Downarrow v_1, \sigma_1 \qquad \rho, \tau \vdash e_2, \sigma_1 \Uparrow \varsigma}{\rho, \tau \vdash e_1(e_2), \sigma \Uparrow \varsigma}\ \langle \lambda x.e', \rho' \rangle = v_1$$

$$\text{(RApp3)}\ \frac{\rho, \tau \vdash e_1, \sigma \Downarrow v_1, \sigma_1 \qquad \rho, \tau \vdash e_2, \sigma_1 \Downarrow v_2, \sigma_2 \qquad \rho'[x \mapsto \ell], \tau' \vdash e', \sigma_2[\ell \mapsto v_2] \Uparrow \varsigma}{\rho, \tau \vdash e_1(e_2), \sigma \Uparrow \varsigma} \begin{array}{r} \langle \lambda x.e', \rho' \rangle = v_1 \\ \ell = \langle x, \tau' \rangle \\ \tau' \quad \textit{fresh} \end{array}$$

$$\text{(RIf1)}\ \frac{\rho, \tau \vdash e_1, \sigma \Uparrow \varsigma}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \sigma \Uparrow \varsigma} \qquad \text{(RIfT)}\ \frac{\rho, \tau \vdash e_1, \sigma \Downarrow n, \sigma_1 \qquad \rho, \tau \vdash e_2, \sigma_1 \Uparrow \varsigma}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \sigma \Uparrow \varsigma}\ n = 0$$

$$\text{(RIfF)}\ \frac{\rho, \tau \vdash e_1, \sigma \Downarrow n, \sigma_1 \qquad \rho, \tau \vdash e_3, \sigma_1 \Uparrow \varsigma}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \sigma \Uparrow \varsigma}\ n \neq 0$$

Figure 13. λIF Big-step Concrete Evaluation and Reachability Semantics

8.2 From Symbolic Execution to Verification

Traditional symbolic executors mainly aim to find bugs and do not provide termination guarantee. However, when we apply to this symbolic executor the finite abstractions presented in previous sections, namely base value widening and finite allocation (Section 3.3), and caching and fixing (Section 4), we turn the symbolic execution into a sound, path-sensitive program verification.

Operations on symbolic values introduce a new source of infinity in the state-space, because the space of symbolic values is not finite. We therefore 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. Try It Out

All of the components discussed in this paper have been implemented as units [11] in Racket [12]. We have also implemented a `#lang` language so that composing and experimenting with these interpreters is easy. Assuming Racket is installed, you can install the `monadic-eval` package with (URL redacted for double-blind).

10. Formalism

In this section we formalize our approach to designing definitional abstract interpreters. We begin with a "ground truth" big-step semantics and concludes with the fixpoint iteration strategy described in Section 4, which we prove sound and computable w.r.t. a synthesized abstract semantics. The design is systematic, and applies to arbitrary developments which use big-step operational semantics. We demonstrate the systematic process as applied to a subset of the language described in Figure 1, which we call λIF :

$$\begin{array}{lll} e \in \mathit{exp} ::= n \mid x \mid \lambda x.e \mid e(e) \mid \texttt{if0}(e)\{e\}\{e\} \mid b(e,e) & & \\ n \in \mathit{nums} & \ell \in \mathit{addr} := \mathit{var} \times \mathit{time} \\ x \in \mathit{vars} & \sigma \in \mathit{store} := \mathit{addr} \to \mathit{val}_\perp \\ b \in \mathit{binop} := \{\mathit{plus}, \ldots\} & v \in \mathit{val} ::= n \mid \langle \lambda x.e, \rho \rangle \\ \tau \in \mathit{time} := \mathbb{N} & \rho \in \mathit{env} := \mathit{var} \to \mathit{addr}_\perp \end{array}$$

Concrete Semantics We begin with the concrete semantics of λIF as a big-step evaluation relation $\rho, \tau \vdash e, \sigma \Downarrow v, \sigma'$, shown in Figure 13. The definition is mostly standard: ρ and σ are the environment and store, e is the initial expression, and v is the resulting value. The argument τ represents "time," which when abstracted supports modeling execution contexts like call-site sensitivity. Concretely time is modelled as a natural number, and all that is required is that "fresh" numbers are available for allocating values in the store.

Reachability The primary limitation of using big-step semantics as a starting point for abstraction is that intermediate computations are not represented in the model for evaluation. For example, consider the program that applies the identity function to an expression that loops, which we notate Ω:

$$(\lambda x.x)(\Omega)$$

A big-step evaluation relation can only describe results of terminating computations, and because this program never terminates, such a relation says nothing about the behavior of the program. A good static analyzer will explore the behavior of Ω to (possibly) discover that it loops, or more importantly, to provide analysis results (like data-flow or side-effects) for intermediate computation states.

The need to analyze intermediate states is the primary reason that big-step semantics are overlooked as a starting point for abstract

[*Collecting Evaluation*] $\boxed{\rho, \tau \vdash e, \widetilde{\sigma} \Downarrow \widetilde{v}, \widetilde{\sigma}'}$

$$\text{(OLit)}\ \frac{}{\rho, \tau \vdash n, \widetilde{\sigma} \Downarrow \{n\}, \widetilde{\sigma}} \qquad \text{(OVar)}\ \frac{}{\rho, \tau \vdash x, \widetilde{\sigma} \Downarrow \widetilde{\sigma}(\rho(x)), \widetilde{\sigma}} \qquad \text{(OLam)}\ \frac{}{\rho, \tau \vdash \lambda x.e, \widetilde{\sigma} \Downarrow \{\langle \lambda x.e, \rho \rangle\}, \widetilde{\sigma}}$$

$$\text{(OBin)}\ \frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Downarrow \widetilde{v}_1, \widetilde{\sigma}_1 \qquad \rho, \tau \vdash e_2, \widetilde{\sigma}_1 \Downarrow \widetilde{v}_2, \widetilde{\sigma}_2}{\rho, \tau \vdash b(e_1, e_2), \widetilde{\sigma} \Downarrow \widetilde{[\![b]\!]}(\widetilde{v}_1, \widetilde{v}_2), \widetilde{\sigma}_2}$$

$$\text{(OApp)}\ \frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Downarrow \widetilde{v}_1, \widetilde{\sigma}_1 \qquad \rho, \tau \vdash e_2, \widetilde{\sigma}_1 \Downarrow \widetilde{v}_2, \widetilde{\sigma}_2 \qquad \rho'[x \mapsto \ell], \tau' \vdash e', \widetilde{\sigma}_2[\ell \mapsto \widetilde{v}_2] \Downarrow \widetilde{v}', \widetilde{\sigma}_3}{\rho, \tau \vdash e_1(e_2), \widetilde{\sigma} \Downarrow \widetilde{v}', \widetilde{\sigma}_3}\ \begin{array}{l}\langle \lambda x.e', \rho' \rangle \in \widetilde{v}_1 \\ \ell = \langle x, \tau' \rangle \\ \tau' \quad \textit{fresh}\end{array}$$

$$\text{(OIfT)}\ \frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Downarrow \widetilde{v}_1, \widetilde{\sigma}_1 \qquad \rho, \tau \vdash e_2, \widetilde{\sigma}_1 \Downarrow \widetilde{v}, \widetilde{\sigma}_2}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \widetilde{\sigma} \Downarrow \widetilde{v}, \widetilde{\sigma}_2}\ 0 \in \widetilde{v}_1 \qquad \text{(OIfF)}\ \frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Downarrow \widetilde{v}_1, \widetilde{\sigma}_1 \qquad \rho, \tau \vdash e_3, \widetilde{\sigma}_1 \Downarrow \widetilde{v}, \widetilde{\sigma}_2}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \widetilde{\sigma} \Downarrow \widetilde{v}, \widetilde{\sigma}_2}\ \begin{array}{l} n \in \widetilde{v}_1 \\ n \neq 0\end{array}$$

[*Collecting Reachability*] $\boxed{\rho, \tau \vdash e, \widetilde{\sigma} \Uparrow \widetilde{\varsigma}}$

$$\text{(ORefl)}\ \frac{}{\rho, \tau \vdash e, \widetilde{\sigma} \Uparrow \langle e, \rho, \widetilde{\sigma}, \tau \rangle} \qquad \text{(ORBin1)}\ \frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Uparrow \varsigma}{\rho, \tau \vdash b(e_1, e_2), \widetilde{\sigma} \Uparrow \widetilde{\varsigma}} \qquad \text{(ORBin2)}\ \frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Downarrow \widetilde{v}_1, \widetilde{\sigma}_1 \qquad \rho, \tau \vdash e_2, \widetilde{\sigma}_1 \Uparrow \widetilde{\varsigma}}{\rho, \tau \vdash b(e_1, e_2), \widetilde{\sigma} \Uparrow \widetilde{\varsigma}}$$

$$\text{(ORApp1)}\ \frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Uparrow \widetilde{\varsigma}}{\rho, \tau \vdash e_1(e_2), \widetilde{\sigma} \Uparrow \widetilde{\varsigma}} \qquad \text{(ORApp2)}\ \frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Downarrow \widetilde{v}_1, \widetilde{\sigma}_1 \qquad \rho, \tau \vdash e_2, \widetilde{\sigma}_1 \Uparrow \widetilde{\varsigma}}{\rho, \tau \vdash e_1(e_2), \widetilde{\sigma} \Uparrow \widetilde{\varsigma}}\ \langle \lambda x.e', \rho' \rangle \in \widetilde{v}_1$$

$$\text{(ORApp3)}\ \frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Downarrow \widetilde{v}_1, \widetilde{\sigma}_1 \qquad \rho, \tau \vdash e_2, \widetilde{\sigma}_1 \Downarrow \widetilde{v}_2, \widetilde{\sigma}_2 \qquad \rho'[x \mapsto \ell], \tau' \vdash e', \widetilde{\sigma}_2[\ell \mapsto \widetilde{v}_2] \Uparrow \widetilde{\varsigma}}{\rho, \tau \vdash e_1(e_2), \widetilde{\sigma} \Uparrow \widetilde{\varsigma}}\ \begin{array}{l}\langle \lambda x.e', \rho' \rangle \in \widetilde{v}_1 \\ \ell = \langle x, \tau' \rangle \\ \tau' \quad \textit{fresh}\end{array}$$

$$\text{(ORIf1)}\ \frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Uparrow \widetilde{\varsigma}}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \widetilde{\sigma} \Uparrow \widetilde{\varsigma}} \qquad \text{(ORIfT)}\ \frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Downarrow \widetilde{v}_1, \widetilde{\sigma}_1 \qquad \rho, \tau \vdash e_2, \widetilde{\sigma}_1 \Uparrow \widetilde{\varsigma}}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \widetilde{\sigma} \Uparrow \widetilde{\varsigma}}\ 0 \in \widetilde{v}_1$$

$$\text{(ORIfF)}\ \frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Downarrow \widetilde{v}_1, \widetilde{\sigma}_1 \qquad \rho, \tau \vdash e_3, \widetilde{\sigma}_1 \Uparrow \widetilde{\varsigma}}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \widetilde{\sigma} \Uparrow \widetilde{\varsigma}}\ \begin{array}{l} n \in \widetilde{v}_1 \\ n \neq 0\end{array}$$

Figure 14. Big-step Collecting Evaluation and Reachability Semantics

interpretation. To remedy the situation, while remaining in a big-step setting, we introduce a big-step *reachability* relation, notated $\rho, \tau \vdash e, \sigma \Uparrow \varsigma$ and also shown in Figure 13. Configurations ς are tuples $\langle e, \rho, \sigma, \tau \rangle$, and are reachable when evaluation passes through the configuration at any point on its way to a final value, or during an infinite loop.

The complete big-step semantics of an expression (e) under environment (ρ), store (σ) and time (τ), which we notate $[\![e]\!]^{bs}(\rho, \sigma, \tau)$, is then the set of all *reachable evaluations*:

$$[\![e]\!]^{bs}(\rho, \sigma, \tau) \coloneqq \{\langle v, \sigma'' \rangle \mid \rho, \sigma, \tau \Uparrow \langle e', \rho', \sigma', \tau' \rangle \wedge \rho', \tau' \vdash e', \sigma' \Downarrow v, \sigma''\}$$

We construct a formal bridge between the big-step and small-step worlds through the complete big-step semantics ($[\![e]\!]^{bs}$) and a complete small-step semantics $\rightsquigarrow^*$, which is traditionally used as the starting point of abstraction for program analysis:

$$[\![e]\!]^{ss}(\rho, \sigma, \tau) \coloneqq \{\langle v, \sigma'' \rangle \mid \forall \kappa. \langle e, \rho, \sigma, \tau, \kappa \rangle \rightsquigarrow^* \langle e', \rho', \sigma', \tau', \kappa' +\!\!+ \kappa \rangle \wedge \langle e', \rho', \sigma', \tau', \kappa' +\!\!+ \kappa \rangle \rightsquigarrow^* \langle v, \rho'', \sigma'', \tau'', \kappa' +\!\!+ \kappa \rangle\}$$

We connect the complete big-step and small-step semantics through the following theorem:

THEOREM 1 (Complete Big-step/Small-step Equivalence).

$$[\![e]\!]^{bs}(\rho, \sigma, \tau) = [\![e]\!]^{ss}(\rho, \sigma, \tau)$$

The proof is by induction on the big-step derivation for $\subseteq$, and on the transitive small-step derivation for $\supseteq$.

Collecting Semantics Before abstracting the semantics—in pursuit of a sound static analysis algorithm—we pass through a big-step collecting evaluation and reachability semantics, notated $\rho, \tau \vdash e, \widetilde{\sigma} \Downarrow \widetilde{v}, \widetilde{\sigma}$ and $\rho, \tau \vdash e, \widetilde{\sigma} \Uparrow \widetilde{\varsigma}$ and shown in Figure 14, where $\widetilde{v}$, $\widetilde{\sigma}$ and $\widetilde{\varsigma}$ range over collecting state spaces:

$$\begin{array}{rll} \widetilde{v} \in & \widetilde{val} & \coloneqq \wp(val) \\ \widetilde{\sigma} \in & \widetilde{store} & \coloneqq addr \mapsto \widetilde{val} \\ \widetilde{\varsigma} \in & \widetilde{config} & \coloneqq exp \times env \times \widetilde{store} \times time \end{array}$$

and the denotation for binary operators ($[\![b]\!]$) is lifted to a collecting denotation operator $\widetilde{[\![b]\!]}$:

$$\widetilde{[\![b]\!]}(\widetilde{v}_1, \widetilde{v}_2) \coloneqq \{[\![b]\!](v_1, v_2) \mid v_1 \in \widetilde{v}_1 \wedge v_2 \in \widetilde{v}_2\}$$

The big-step collecting and reachability relations are structurally similar to the concrete semantics. The primary differences are the use of set containment ($\in$) in place of equality ($=$) when branching on application and conditional expressions.

The big-step collecting reachability semantics is a sound approximation of the big-step concrete reachability semantics:

THEOREM 2 (Collecting Reachability Semantics Soundness).

If $\rho, \tau \vdash e, \sigma \Uparrow \langle e', \rho', \sigma', \tau' \rangle$ *and* $\rho', \tau' \vdash e', \sigma' \Downarrow v, \sigma''$
where $\eta(\sigma) \sqsubseteq \widetilde{\sigma}$
then $\rho, \tau \vdash e, \widetilde{\sigma} \Uparrow \langle e', \rho', \widetilde{\sigma}', \tau' \rangle$ *and* $\rho', \tau' \vdash e, \widetilde{\sigma}' \Downarrow \widetilde{v}, \widetilde{\sigma}''$
where $\eta(\sigma') \sqsubseteq \widetilde{\sigma}'$ *and* $v \in \widetilde{v}$ *and* $\eta(\sigma'') \sqsubseteq \widetilde{\sigma}''$

The proof is by induction on the concrete big-step derivation. The extraction function η is defined separately for stores (σ) and configurations (ς):

$$\eta(\sigma)(\ell) \coloneqq \{\sigma(\ell)\} \qquad \eta(\langle e, \rho, \sigma, \tau \rangle) \coloneqq \langle e, \rho, \eta(\sigma), \tau \rangle$$

and the partial ordering on stores and configurations is pointwise:

$$\begin{array}{l} \widetilde{\sigma}_1 \sqsubseteq \widetilde{\sigma}_2 \quad \textit{iff} \quad \forall \ell.\ \widetilde{\sigma}_1(\ell) \subseteq \widetilde{\sigma}_2(\ell) \\ \langle e_1, \rho_1, \widetilde{\sigma}_1, \tau_1 \rangle \sqsubseteq \langle e_2, \rho_2, \widetilde{\sigma}_2, \tau_2 \rangle \quad \textit{iff} \\ \qquad e_1 = e_2 \wedge \rho_1 = \rho_2 \wedge \widetilde{\sigma}_1 \sqsubseteq \widetilde{\sigma}_2 \wedge \tau_1 = \tau_2 \end{array}$$

[*Abstract Evaluation*] $\boxed{\widehat{\rho},\widehat{\tau}\vdash e,\widehat{\sigma}\Downarrow\widehat{v},\widehat{\sigma}'}$

$$\text{(ALIT)}\ \frac{}{\widehat{\rho},\widehat{\tau}\vdash n,\widehat{\sigma}\Downarrow\widehat{\eta}(n),\widehat{\sigma}} \qquad \text{(AVAR)}\ \frac{}{\widehat{\rho},\widehat{\tau}\vdash x,\widehat{\sigma}\Downarrow\widehat{\sigma}(\widehat{\rho}(x)),\widehat{\sigma}} \qquad \text{(ALAM)}\ \frac{}{\widehat{\rho},\widehat{\tau}\vdash \lambda x.e,\widehat{\sigma}\Downarrow\widehat{\eta}(\langle\lambda x.e,\widehat{\rho}\rangle),\widehat{\sigma}}$$

$$\text{(ABIN)}\ \frac{\widehat{\rho},\widehat{\tau}\vdash e_1,\widehat{\sigma}\Downarrow\widehat{v}_1,\widehat{\sigma}_1 \qquad \widehat{\rho},\widehat{\tau}\vdash e_2,\widehat{\sigma}_1\Downarrow\widehat{v}_2,\widehat{\sigma}_2}{\widehat{\rho},\widehat{\tau}\vdash b(e_1,e_2),\widehat{\sigma}\Downarrow\widehat{[\![b]\!]}(\widehat{v}_1,\widehat{v}_2),\widehat{\sigma}_2}$$

$$\text{(AAPP)}\ \frac{\widehat{\rho},\widehat{\tau}\vdash e_1,\widehat{\sigma}\Downarrow\widehat{v}_1,\widehat{\sigma}_1 \qquad \widehat{\rho},\widehat{\tau}\vdash e_2,\widehat{\sigma}_1\Downarrow\widehat{v}_2,\widehat{\sigma}_2 \qquad \widehat{\rho}'[x\mapsto\widehat{\ell}],\widehat{\tau}'\vdash e',\widehat{\sigma}_2\sqcup[\widehat{\ell}\mapsto\widehat{v}_2]\Downarrow\widehat{v}',\widehat{\sigma}_3}{\widehat{\rho},\widehat{\tau}\vdash e_1(e_2),\widehat{\sigma}\Downarrow\widehat{v}',\widehat{\sigma}_3}\ \begin{array}{l}\langle\lambda x.e',\widehat{\rho}'\rangle\in\lfloor\gamma\rfloor_{clo}(\widehat{v}_1)\\ \widehat{\varsigma}=\langle e_1(e_2),\widehat{\rho},\widehat{\sigma},\widehat{\tau}\rangle\\ \widehat{\ell}=\langle x,\widehat{\tau}'\rangle\\ \widehat{\tau}'=\widehat{next}(\widehat{\tau},\widehat{\varsigma})\end{array}$$

$$\text{(AIFT)}\ \frac{\widehat{\rho},\widehat{\tau}\vdash e_1,\widehat{\sigma}\Downarrow\widehat{v}_1,\widehat{\sigma}_1 \qquad \widehat{\rho},\widehat{\tau}\vdash e_2,\widehat{\sigma}_1\Downarrow\widehat{v},\widehat{\sigma}_2}{\widehat{\rho},\widehat{\tau}\vdash \texttt{if0}(e_1)\{e_2\}\{e_3\},\widehat{\sigma}\Downarrow\widehat{v},\widehat{\sigma}_2}\ 0\in\lfloor\gamma\rfloor_0(\widehat{v}_1) \qquad \text{(AIFF)}\ \frac{\widehat{\rho},\widehat{\tau}\vdash e_1,\widehat{\sigma}\Downarrow\widehat{v}_1,\widehat{\sigma}_1 \qquad \widehat{\rho},\widehat{\tau}\vdash e_3,\widehat{\sigma}_1\Downarrow\widehat{v},\widehat{\sigma}_2}{\widehat{\rho},\widehat{\tau}\vdash \texttt{if0}(e_1)\{e_2\}\{e_3\},\widehat{\sigma}\Downarrow\widehat{v},\widehat{\sigma}_2}\ \neg 0\in\lfloor\gamma\rfloor_{\neg 0}(\widehat{v}_1)$$

[*Abstract Reachability*] $\boxed{\widehat{\rho},\widehat{\tau}\vdash e,\widehat{\sigma}\Uparrow\widehat{\varsigma}}$

$$\text{(AREFL)}\ \frac{}{\widehat{\rho},\widehat{\tau}\vdash e,\widehat{\sigma}\Uparrow\langle e,\widehat{\rho},\widehat{\sigma},\widehat{\tau}\rangle} \qquad \text{(ARBIN1)}\ \frac{\widehat{\rho},\widehat{\tau}\vdash e_1,\widehat{\sigma}\Uparrow\varsigma}{\widehat{\rho},\widehat{\tau}\vdash b(e_1,e_2),\widehat{\sigma}\Uparrow\widehat{\varsigma}} \qquad \text{(ARBIN2)}\ \frac{\widehat{\rho},\widehat{\tau}\vdash e_1,\widehat{\sigma}\Downarrow\widehat{v}_1,\widehat{\sigma}_1 \qquad \widehat{\rho},\widehat{\tau}\vdash e_2,\widehat{\sigma}_1\Uparrow\widehat{\varsigma}}{\widehat{\rho},\widehat{\tau}\vdash b(e_1,e_2),\widehat{\sigma}\Uparrow\widehat{\varsigma}}$$

$$\text{(ARAPP1)}\ \frac{\widehat{\rho},\widehat{\tau}\vdash e_1,\widehat{\sigma}\Uparrow\widehat{\varsigma}}{\widehat{\rho},\widehat{\tau}\vdash e_1(e_2),\widehat{\sigma}\Uparrow\widehat{\varsigma}} \qquad \text{(ARAPP2)}\ \frac{\widehat{\rho},\widehat{\tau}\vdash e_1,\widehat{\sigma}\Downarrow\widehat{v}_1,\widehat{\sigma}_1 \qquad \widehat{\rho},\widehat{\tau}\vdash e_2,\widehat{\sigma}_1\Uparrow\widehat{\varsigma}}{\widehat{\rho},\widehat{\tau}\vdash e_1(e_2),\widehat{\sigma}\Uparrow\widehat{\varsigma}}\ \langle\lambda x.e',\widehat{\rho}'\rangle\in\lfloor\gamma\rfloor_{clo}(\widehat{v}_1)$$

$$\text{(ARAPP3)}\ \frac{\widehat{\rho},\widehat{\tau}\vdash e_1,\widehat{\sigma}\Downarrow\widehat{v}_1,\widehat{\sigma}_1 \qquad \widehat{\rho},\widehat{\tau}\vdash e_2,\widehat{\sigma}_1\Downarrow\widehat{v}_2,\widehat{\sigma}_2 \qquad \widehat{\rho}'[x\mapsto\widehat{\ell}],\widehat{\tau}'\vdash e',\widehat{\sigma}_2\sqcup[\widehat{\ell}\mapsto\widehat{v}_2]\Uparrow\widehat{\varsigma}}{\widehat{\rho},\widehat{\tau}\vdash e_1(e_2),\widehat{\sigma}\Uparrow\widehat{\varsigma}}\ \begin{array}{l}\langle\lambda x.e',\widehat{\rho}'\rangle\in\lfloor\gamma\rfloor_{clo}(\widehat{v}_1)\\ \widehat{\varsigma}=\langle e_1(e_2),\widehat{\rho},\widehat{\sigma},\widehat{\tau}\rangle\\ \widehat{\ell}=\langle x,\widehat{\tau}'\rangle\\ \widehat{\tau}'=\widehat{next}(\widehat{\tau},\widehat{\varsigma})\end{array}$$

$$\text{(ARIF1)}\ \frac{\widehat{\rho},\widehat{\tau}\vdash e_1,\widehat{\sigma}\Uparrow\widehat{\varsigma}}{\widehat{\rho},\widehat{\tau}\vdash \texttt{if0}(e_1)\{e_2\}\{e_3\},\widehat{\sigma}\Uparrow\widehat{\varsigma}} \qquad \text{(ARIFT)}\ \frac{\widehat{\rho},\widehat{\tau}\vdash e_1,\widehat{\sigma}\Downarrow\widehat{v}_1,\widehat{\sigma}_1 \qquad \widehat{\rho},\widehat{\tau}\vdash e_2,\widehat{\sigma}_1\Uparrow\widehat{\varsigma}}{\widehat{\rho},\widehat{\tau}\vdash \texttt{if0}(e_1)\{e_2\}\{e_3\},\widehat{\sigma}\Uparrow\widehat{\varsigma}}\ 0\in\lfloor\gamma\rfloor_0(\widehat{v}_1)$$

$$\text{(ARIFF)}\ \frac{\widehat{\rho},\widehat{\tau}\vdash e_1,\widehat{\sigma}\Downarrow\widehat{v}_1,\widehat{\sigma}_1 \qquad \widehat{\rho},\widehat{\tau}\vdash e_3,\widehat{\sigma}_1\Uparrow\widehat{\varsigma}}{\widehat{\rho},\widehat{\tau}\vdash \texttt{if0}(e_1)\{e_2\}\{e_3\},\widehat{\sigma}\Uparrow\widehat{\varsigma}}\ \neg 0\in\lfloor\gamma\rfloor_{\neg 0}(\widehat{v}_1)$$

Figure 15. Big-step Abstract Evaluation and Reachability Semantics

Finite Abstraction The next step towards a computable static analysis is an abstract semantics with a finite state space that approximates the big-step collecting semantics, notated $\widehat{\rho},\widehat{\tau}\vdash e,\widehat{\sigma}\Downarrow\widehat{v},\widehat{\sigma}$ and $\widehat{\rho},\widehat{\tau}\vdash e,\widehat{\sigma}\Uparrow\widehat{\varsigma}$ and shown in Figure 15, where $\widehat{\rho}$, $\widehat{\tau}$, $\widehat{v}$, $\widehat{\sigma}$ and $\widehat{\varsigma}$ are finite abstractions of their collecting counterparts:

$$\begin{array}{lll}\widehat{\rho}\in & \widehat{env}\coloneqq var\mapsto\widehat{addr}_\perp\\ \widehat{\ell}\in & \widehat{addr}\coloneqq var\times\widehat{time}\\ \widehat{\tau}\in & \widehat{time}\coloneqq\ldots\\ \widehat{v}\in & \widehat{val}\coloneqq\ldots\\ \widehat{\sigma}\in & \widehat{store}\coloneqq\widehat{addr}\mapsto\widehat{val}\\ \widehat{\varsigma}\in & \widehat{config}\coloneqq exp\times\widehat{env}\times\widehat{store}\times\widehat{time}\end{array}$$

The primary structural difference from the collecting semantics is the use of join when updating the store ($\widehat{\sigma}\sqcup[\widehat{\ell}\mapsto\widehat{v}]$) rather than strict replacement ($\widetilde{\sigma}[\ell\mapsto\widetilde{v}]$). This is to preserve soundness in the presence of address reuse, which occurs from the finite size of the address space.

The abstract denotation ($\widehat{[\![b]\!]}$) is any over-approximation of the collecting denotation ($\widetilde{[\![b]\!]}$) w.r.t. a Galois connection $\widetilde{val}\ \underset{\alpha}{\overset{\gamma}{\leftrightarrows}}\ \widehat{val}$:

$$\widehat{[\![b]\!]}(\widehat{v}_1,\widehat{v}_2)\sqsupseteq\alpha(\widetilde{[\![b]\!]}(\gamma(\widehat{v}_1),\gamma(\widehat{v}_2)))$$

Concretization functions $\lfloor\gamma\rfloor_{clo}$, $\lfloor\gamma\rfloor_0$ and $\lfloor\gamma\rfloor_{\neg 0}$ are computable finite subsets of the full concretization function γ s.t.:

$$\begin{array}{rl}\lfloor\gamma\rfloor_{clo}(\widehat{v})\coloneqq & \{\langle\lambda x.e,\widehat{\rho}\rangle\mid\langle\lambda x.e,\widehat{\rho}\rangle\in\gamma(\widehat{v})\}\\ \lfloor\gamma\rfloor_0(\widehat{v})\coloneqq & \{0\mid 0\in\gamma(\widehat{v})\}\\ \lfloor\gamma\rfloor_{\neg 0}(\widehat{v})\coloneqq & \{\neg 0\mid n\in\gamma(\widehat{v})\wedge n\neq 0\}\end{array}$$

Abstract sets $\widehat{time}$ and $\widehat{val}$ are left as parameters to the analysis along with their operations $\widehat{next}$, $\widehat{[\![b]\!]}$, $\lfloor\gamma\rfloor_{clo}$, $\lfloor\gamma\rfloor_0$, $\lfloor\gamma\rfloor_{\neg 0}$ and $\sqcup^{\widehat{val}}$.

The abstract semantics is a sound approximation of the collecting semantics, which we establish through the theorem:

THEOREM 3 (Abstract Reachability Semantics Soundness).

$$\begin{array}{lll}\textit{If} & \rho,\tau\vdash e,\widetilde{\sigma}\Uparrow\langle e',\rho',\widetilde{\sigma}',\tau'\rangle & \textit{and}\quad \rho',\tau'\vdash e',\widetilde{\sigma}'\Downarrow\widetilde{v},\widetilde{\sigma}''\\ \textit{where} & \eta(\rho)\sqsubseteq\widehat{\rho} \quad \textit{and}\quad \eta(\tau)\sqsubseteq\widehat{\tau} & \textit{and}\quad \eta(\widetilde{\sigma})\sqsubseteq\widehat{\sigma}\\ \textit{then} & \widehat{\rho},\widehat{\tau}\vdash e,\widehat{\sigma}\Uparrow\langle e',\widehat{\rho}',\widehat{\sigma}',\widehat{\tau}'\rangle & \textit{and}\quad \widehat{\rho}',\widehat{\tau}'\vdash e,\widehat{\sigma}'\Downarrow\widehat{v},\widehat{\sigma}''\\ \textit{where} & \eta(\rho')\sqsubseteq\widehat{\rho}',\eta(\tau')\sqsubseteq\widehat{\tau}',\eta(\widetilde{\sigma}')\sqsubseteq\widehat{\sigma}',v\in\widetilde{v},\eta(\sigma'')\sqsubseteq\widetilde{\sigma}''\end{array}$$

The proof is by induction on the big-step derivation. The extraction function η is defined separately for environments (ρ), time (τ), collecting stores ($\widetilde{\sigma}$), values ($\widetilde{v}$) and configurations ($\widetilde{\varsigma}$). $\eta(\tau)$ and $\eta(\widetilde{v})$ are given with parameters $\widehat{time}$ and $\widehat{val}$. $\eta(\rho)$, $\eta(\widetilde{\sigma})$ and $\eta(\widetilde{\varsigma})$ are defined pointwise:

$$\eta(\rho)(x)\coloneqq\eta(\rho(x)) \qquad \eta(\widetilde{\sigma})(\widehat{\ell})\coloneqq\bigsqcup_{\ell\in\gamma(\widehat{\ell})}\eta(\widetilde{\sigma}(\ell))$$
$$\eta(\langle e,\rho,\tau,\widetilde{\sigma}\rangle)\coloneqq\langle e,\eta(\rho),\eta(\tau),\eta(\widetilde{\sigma})\rangle$$

Computing the Analysis An analysis for the program e_0 w.r.t. the abstract semantics is some cache $\$\in\widehat{config}\mapsto\wp(\widehat{val}\times\widehat{store})$ that maps all configurations reachable from the initial configuration $\langle e_0,\widehat{\rho}_0,\widehat{\sigma}_0,\widehat{\tau}_0\rangle$ to their final values and stores $\widehat{v},\widehat{\sigma}$, which we notate $\$\models e_0$:

$$\begin{array}{lll}\$\models e_0 & \textit{iff} & \textit{If}\quad \widehat{\rho}_0,\widehat{\tau}_0\vdash e_0,\widehat{\sigma_0}\Uparrow\langle e,\widehat{\rho},\widehat{\sigma},\widehat{\tau}\rangle\\ & & \textit{and}\quad \widehat{\rho},\widehat{\tau}\vdash e,\widehat{\sigma}\Downarrow\widehat{v},\widehat{\sigma}'\\ & & \textit{then}\quad \langle\widehat{v},\widehat{\sigma}'\rangle\in\$(\langle e,\widehat{\rho},\widehat{\sigma},\widehat{\tau}\rangle)\end{array}$$

The *best* cache $\$^+$ is then computed as the least fixed point of the functional $\mathcal{F}$:

$$\mathcal{F} \in (\widehat{config} \mapsto \wp(\widehat{val} \times \widehat{store})) \to (\widehat{config} \mapsto \wp(\widehat{val} \times \widehat{store}))$$
$$\mathcal{F} \coloneqq \lambda\$.\ \bigsqcup_{\langle e,\widehat{\rho},\widehat{\sigma},\widehat{\tau}\rangle \in \$} \begin{cases} \{\langle e,\widehat{\rho},\widehat{\sigma},\widehat{\tau}\rangle \mapsto \{\langle \widehat{v},\widehat{\sigma}'\rangle\} \mid \widehat{\rho},\widehat{\tau} \vdash e,\widehat{\sigma} \Downarrow^{\$} \widehat{v},\widehat{\sigma}'\} \\ \{\widehat{\varsigma} \mapsto \{\} \mid \widehat{\rho},\widehat{\tau} \vdash e,\widehat{\sigma} \Uparrow^{\$} \widehat{\varsigma}\} \end{cases}$$

which also includes the initial configuration:

$$\$^+ \coloneqq \mathit{lfp}(\lambda\$.\ \mathcal{F}(\$) \sqcup \{\langle e_0, \eta(\rho_0), \eta(\sigma_0), \eta(\tau_0)\rangle \mapsto \{\}\})$$

The relations $\widehat{\rho},\widehat{\tau} \vdash e,\widehat{\sigma} \Downarrow^{\$} \widehat{v},\widehat{\sigma}'$ and $\widehat{\rho},\widehat{\tau} \vdash e,\widehat{\sigma} \Uparrow^{\$} \widehat{\varsigma}$ are modified versions of the original abstract semantics, but with recursive judgements replaced by $\langle \widehat{v},\widehat{\sigma}'\rangle \in \$(e,\widehat{\rho},\widehat{\sigma},\widehat{\tau})$ and $\widehat{\varsigma} \in \$(e,\widehat{\rho},\widehat{\sigma},\widehat{\tau})$ respectively. Therefore $\mathcal{F}$ is not recursive; the recursion in the relations is lifted to the outer fixpoint of the analysis. Because the state space $\widehat{config} \mapsto \wp(\widehat{val} \times \widehat{store})$ is finite and $\mathcal{F}$ is monotonic, $\$^+$ can be computed algorithmically in finite time by Kleene fixed-point iteration. See Nielson et al [28] for more background and examples of static analyzers computed in this style, and from which the current development was largely inspired.

THEOREM 4 (Algorithm Correctness). $\$^+$ *is a valid analysis for* e_0*, that is:* $\$^+ \models e_0$.

The proof is by induction on the assumed derivations $\widehat{\rho}_0,\widehat{\tau}_0 \vdash e_0,\widehat{\sigma}_0 \Uparrow \langle \widehat{e},\widehat{\rho},\widehat{\sigma},\widehat{\tau}\rangle$ and $\widehat{\rho},\widehat{\tau} \vdash e,\widehat{\sigma} \Downarrow \widehat{v},\widehat{\sigma}'$, and utilizes the fact that $\$^+$ is a fixed point, that is: $\mathcal{F}(\$^+) = \$^+$. Our final theorem relates the analysis cache $\$^+$ back to the concrete semantics of the initial program as a sound approximation:

THEOREM 5 (Algorithm Soundness).

If	$\rho_0,\tau_0 \vdash e_0,\sigma_0 \Uparrow \langle e,\rho,\sigma,\tau\rangle$	*and*	$\rho,\tau \vdash e,\sigma \Downarrow v,\sigma'$
then	$\langle \widehat{v},\widehat{\sigma}'\rangle \in \$^+(\langle e,\widehat{\rho},\widehat{\sigma},\widehat{\tau}\rangle)$		
where	$\eta(\rho) \sqsubseteq \widehat{\rho}$, $\eta(\tau) \sqsubseteq \widehat{\tau}$, $\eta(\sigma) \sqsubseteq \widehat{\sigma}$, $\eta(v) \sqsubseteq \widehat{v}$, $\eta(\sigma') \sqsubseteq \widehat{\sigma}'$		

The proof follows by composing Theorems 1-4.

Computing with Definitional Interpreters The algorithm described in Section 4 is a more efficient strategy for computing $\$^+$ using an extensible open-recursive definitional interpreter. This technique is general, and bridges the gap between the big-step abstract semantics formalized in this section and the definitional interpreters we wish to execute to obtain analyses.

An extensible open-recursive definitional interpreter for λIF (the small language formalized in this section) has domain:

$$\mathcal{E} \in \Sigma \to \Sigma \quad \textit{where} \quad \Sigma \coloneqq \widehat{config} \to \wp(\widehat{val} \times \widehat{store})$$

and is defined such that its denotational-fixpoint ($Y(\mathcal{E})$) recovers concrete interpretation when instantiated with the concrete state-space. For example, the recursive case for binary operator expressions is defined:

$$\begin{aligned}&\mathcal{E}(\mathcal{E}')(\langle b(e_1,e_2),\widehat{\rho},\widehat{\sigma},\widehat{\tau}\rangle \coloneqq \\ &\quad \{\ \widehat{[\![b]\!]}(\widehat{v}_1,\widehat{v}_2) \\ &\quad \mid \langle \widehat{v}_1,\widehat{\sigma}_1\rangle \in \mathcal{E}'(\langle e_1,\widehat{\rho},\widehat{\sigma},\widehat{\tau}\rangle) \wedge \langle \widehat{v}_2,\widehat{\sigma}_2\rangle \in \mathcal{E}'(\langle e_2,\widehat{\rho},\widehat{\sigma}_1,\widehat{\tau}\rangle)\}\end{aligned}$$

The iteration strategy to analyze the program e_0 is then to run e_0 using $\mathcal{E}$, but intercepting recursive calls to:

1. Cache results for all intermediate configurations $\widehat{\varsigma}$; and
2. Cache seen states to prevent infinite loops.

(1) is required to fulfill the specification that $\$^+$ include results for all reachable configurations from e_0, and (2) is required to reach a fixed point of the analysis. To track this extra information we add functional state to the interpreter (which was done through a monad transformer in Section 4) of type:

$$\widehat{cache} \coloneqq \widehat{config} \mapsto \wp(\widehat{val} \times \widehat{store})$$

such that the open-recursive evaluator has type:

$$\begin{aligned}&\mathcal{E} \in \Sigma \to \Sigma \quad \textit{where} \\ &\quad \Sigma \coloneqq \widehat{config} \times \widehat{cache} \to \wp(\widehat{val} \times \widehat{store}) \times \widehat{cache}\end{aligned}$$

The iteration to compute $\$^+$ given $\mathcal{E}$ is then defined:

$$\begin{aligned}&\$^+ \coloneqq \mathit{lfp}(\lambda\$^o. \\ &\quad \texttt{let}\ \mathcal{E}^* \coloneqq Y(\lambda\mathcal{E}'.\ \mathcal{E}(\lambda\langle\widehat{\varsigma},\$^i\rangle. \\ &\qquad \texttt{if}\ \widehat{\varsigma} \in \$^i\ \texttt{then}\ \langle \$^i(\widehat{\varsigma}),\$^i\rangle\ \texttt{else} \\ &\qquad \texttt{let}\ \langle \widehat{VS},\$^{i\prime}\rangle \coloneqq \mathcal{E}'(\widehat{\varsigma},\$^i[\widehat{\varsigma} \mapsto \$^o(\widehat{\varsigma})]) \\ &\qquad \texttt{in}\ \langle \widehat{VS},\$^{i\prime}[\widehat{\varsigma} \mapsto \widehat{VS}]\rangle))) \\ &\quad \texttt{in}\ \pi_2(\mathcal{E}^*(\langle e_0,\widehat{\rho}_0,\widehat{\sigma}_0,\widehat{\tau}_0\rangle,\{\})))\end{aligned}$$

The fixed interpreter $\mathcal{E}^*$ calls the unfixed interpreter $\mathcal{E}$, but intercepts recursive calls to perform (1) and (2) described above. When loops are detected, the results from the previous complete result $\$^o$ is used, and the outer fixpoint computes the least fixed point of this $\$^o$.

The end result is that, rather than compute analysis results and reachable states naively with Kleene fixpoint iteration, we are able to reuse the standard definitional interpreter—written in open-recursive form—to simultaneously explore reachable states, cache intermediate configurations, and iterate towards a least fixpoint solution for the analysis. This method is more efficient, and reuses an extensible definitional interpreter which can recover a wide range of analyses, including concrete interpretation.

Widening Two forms of widening can be employed to the semantics and iteration algorithm to achieve acceptable performance for the abstract interpreter.

The first form of widening is to widen the store in the result set $\wp(\widehat{val} \times \widehat{store})$ to $\wp(\widehat{val}) \times \widehat{store}$ in the evaluator $\mathcal{E}$:

$$\begin{aligned}&\mathcal{E} \in \Sigma \to \Sigma \quad \textit{where} \\ &\quad \Sigma \coloneqq \widehat{config} \times \widehat{cache} \to \wp(\widehat{val}) \times \widehat{store} \times \widehat{cache}\end{aligned}$$

We perform this widening systematically and with no added effort through the use of Galois Transformers [4] in Section 6. The iteration strategy for this widened state space is the same as before, which computes a fixed point of the outer cache $\$^o$.

The next form of widening is to pull the store out of the configuration space *entirely*, that is:

$$\begin{aligned}\widehat{\varsigma} \in\ & \widehat{config} \coloneqq exp \times \widehat{env} \times \widehat{time} \\ \$ \in\ & \widehat{cache} \coloneqq \widehat{config} \mapsto \wp(\widehat{val})\end{aligned}$$

and:

$$\begin{aligned}&\mathcal{E} \in \Sigma \to \Sigma \quad \textit{where} \\ &\quad \Sigma \coloneqq \widehat{config} \times \widehat{store} \times \widehat{cache} \to \wp(\widehat{val}) \times \widehat{store} \times \widehat{cache}\end{aligned}$$

The fixed point iteration then finds a mutual least fixed-point of both the outer cache $\$^o$ *and* the store $\widehat{\sigma}$:

$$\begin{aligned}&\langle \$^+,\widehat{\sigma}^+\rangle \coloneqq \mathit{lfp}(\lambda\langle\$^o,\widehat{\sigma}\rangle. \\ &\quad \texttt{let}\ \mathcal{E}^* \coloneqq Y(\lambda\mathcal{E}'.\ \mathcal{E}(\lambda\langle\widehat{\varsigma},\widehat{\sigma}^i,\$^i\rangle. \\ &\qquad \texttt{if}\ \widehat{\varsigma} \in \$^i\ \texttt{then}\ \langle \$^i(\widehat{\varsigma}),\sigma^i,\$^i\rangle\ \texttt{else} \\ &\qquad \texttt{let}\ \langle \widehat{V},\widehat{\sigma}^{i\prime},\$^{i\prime}\rangle \coloneqq \mathcal{E}'(\widehat{\varsigma},\widehat{\sigma}^i,\$^i[\widehat{\varsigma} \mapsto \$^o(\widehat{\varsigma})]) \\ &\qquad \texttt{in}\ \langle \widehat{V},\widehat{\sigma}^{i\prime},\$^{i\prime}[\widehat{\varsigma} \mapsto \widehat{V}]\rangle))) \\ &\quad \texttt{in}\ \pi_{2\times 3}(\mathcal{E}^*(\langle e_0,\widehat{\rho}_0,\widehat{\tau}_0\rangle,\widehat{\sigma},\{})))\end{aligned}$$

This second version of widening, which computes a fixpoint also over the store, recovers a so-called *flow-insensitive* analysis. In this model, all program states are re-analyzed in the store resulting from

execution. Also, the cache ($) does not index over store states $\widehat{\sigma}$ in its domain, greatly reducing its size, and leading to a much more efficient (although less precise) static analyzer.

Recovering Classical 0CFA From the fully widened static analyzer, which computes a mutual fixpoint between a cache and store, we can easily recover a classical 0CFA analysis. We do this by instantiating $\widehat{time}$ to the singleton abstraction $\{\bullet\}$, as was shown in Section 3. In this setting, the lexical environment ρ is uniquely determined by the program expression e, and can therefore be eliminated, resulting in the analysis state space:

$$\begin{aligned}\widehat{\varsigma} &\in \widehat{config} \coloneqq exp\\ \$ &\in \widehat{cache} \coloneqq exp \mapsto \wp(\widehat{val})\\ \widehat{\sigma} &\in \widehat{store} \coloneqq var \mapsto \wp(\widehat{val})\end{aligned}$$

The specification for the analysis and the fully store-widened least fixed-point iteration for computing it recovers the constraint-based description of 0CFA given by Nielson *et al* in [28], where 0CFA is defined as the smallest cache ($) and store ($\sigma$) which satisfy a co-inductively defined judgment: $\$, \sigma \models e$.

Recovering Pushdown Analysis We borrow from the recent result in pushdown analysis by Gilray *et al* [13] which shows that full pushdown precision can be achieved in a small-step store-widened abstract semantics by allocating continuations using a particular address space: program expressions paired with abstract environments ($\langle e, \widehat{\rho}\rangle$). In other words, $\langle e, \widehat{\rho}\rangle$ is sufficient to achieve full pushdown precision because the tuple uniquely identifies the evaluation context up to the final result of evaluation.

Our fully widened semantics recovers pushdown precision because the cache maps tuples $\langle e, \widehat{\rho}, \widehat{\tau}\rangle$, which contains $\langle e, \widehat{\rho}\rangle$. We then see that abstract time $\widehat{\tau}$ is redundant and eliminate it from the cache, resulting in a smaller domain for the same analysis:

$$\begin{aligned}\widehat{\varsigma} &\in \widehat{config} \coloneqq exp \times \widehat{env} \times \widehat{time}\\ \$ &\in \widehat{cache} \coloneqq exp \times \widehat{env} \mapsto \wp(\widehat{val})\\ \widehat{\sigma} &\in \widehat{store} \coloneqq var \times \widehat{addr} \mapsto \wp(\widehat{val})\end{aligned}$$

An advantage of our setting is that we recover pushdown analysis also for varying degrees of store-widening, which is not the case in Gilray *et al* [13], although pushdown precision for non-widened semantics has been achieved by Johnson and Van Horn [18]. Furthermore, the implementation of our analyzer inherits this precision through precise call-return matching in the defining metalanguage, requiring no added instrumentation to the state-space of the analyzer.

Going back to Nielson et al [28], it would be interesting to redevelop their constraint-based analysis descriptions of kCFA in a form that recovers pushdown precision. Such an exercise would amount to translating our big-step abstract semantics instantiated to kCFA to a constraint system. The resulting system would differ from classical kCFA by the addition of environments $\widehat{\rho}$ (which Nielson et al call context environments) to the domain of the cache. In this way our formal framework is able to bridge the gap between results in pushdown analysis described via small-step machines *a la* Van Horn and Might [38], and constraint-based systems *a la* Nielson et al for which pushdown analysis has yet to be described effectively.

11. Related Work

This work draws upon and re-presents many ideas from the literature on abstract interpretation for higher-order languages. In particular, it closely follows the abstracting abstract machines [38, 39] 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. This involved a novel caching mechanism and fixed-point algorithm, but otherwise followed the same recipe. Remarkably, the pushdown property is simply inherited from the meta-language rather than require explicit mechanisms within the abstract interpreter.

The use of monads and monad transformers to make extensible (concrete) interpreters is a well-known idea [23, 26, 32], which we have extended to work for compositional abstract interpreters. The use of monads and monad transformers in machine based-formulatons of abstract interpreters has previously been explored by Sergey, *et al.* [31] and Darais *et al.* [4], respectively. Darais has also shown that certain monad transformers are also *Galois transformers*, i.e. they compose to form monads that are Galois connections. This idea may pave a path forward for having both componential 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 Johnson and Van Horn [18]. 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 [14].

Caching recursive, non-deterministic functions is a well-studied problem in the functional logic programming community (there termed "tabling") [2, 3, 33, 34], and has been usefully applied to program verification and analysis [5, 16]. 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* [10, 15, 22], and recent work [37, 40] 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 [42], formalized CFA2 using a CPS model, which is similar in spirit to a machine-based model. However, in his dissertation [41] 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 don't believe Big CFA2 as stated is unsound, however if the underlying abstractions were tightened, it may then require a more involved fixpoint 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 [6, 7, 13, 18, 19, 39, 41, 42]. The mixing of symbolic execution and abstract intrepretation is similar in spirit to the *logic flow analysis* of Might [25], albeit in a pushdown setting and with a stronger notion of negation; generally, our presentation resembles traditional formulations of symbolic execution more closely. Our approach to symbolic execution only handles the first-order case of symbolic values, as is traditional. However, Nguyễn's work on higher-order symbolic execution [27] 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.

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 [1]. 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.

12. Conclusions

We have shown that a definitional interpreter written in monadic style can express a wide variety of semantics, such as the usual concrete semantics, collecting semantics, abstract interpretations, symbolic execution, and several combinations thereof.

Remarkably, our abstract interpreter implements a form of pushdown 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 defining language.

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.

Acknowledgments

We thank Sam Tobin-Hochstadt and Dionna Glaze for several fruitful conversations while developing the ideas in this work.

References

[1] M. S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 2005.

[2] R. N. Bol and L. Degerstedt. Tabulated resolution for well founded semantics. In *ILPS*, pages 199–219. Citeseer, 1993.

[3] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM (JACM)*, 43(1):20–74, 1996.

[4] D. Darais, M. Might, and D. V. Horn. Galois transformers and modular abstract interpreters: Reusable metatheory for program analysis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015. ACM, 2015.

[5] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems—a case study. In *ACM SIGPLAN Notices*, volume 31, pages 117–126. ACM, 1996.

[6] C. Earl, M. Might, and D. V. Horn. Pushdown control-flow analysis of higher-order programs. In *Workshop on Scheme and Functional Programming*, 2010.

[7] C. Earl, I. Sergey, M. Might, and D. Van Horn. Introspective pushdown analysis of higher-order programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12. ACM, 2012.

[8] M. Felleisen and D. P. Friedman. A calculus for assignments in higher-order languages. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87. ACM, 1987.

[9] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, Sept. 1992.

[10] S. Fischer, O. Kiselyov, and C. chieh Shan. Purely functional lazy nondeterministic programming. *Journal of Functional Programming*, 21(4-5):413–465, 2011.

[11] M. Flatt and M. Felleisen. Units: Cool modules for hot languages. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98. ACM, 1998.

[12] M. Flatt and PLT. Reference: Racket. Technical report, PLT Inc., 2010.

[13] T. Gilray, S. Lyde, M. D. Adams, M. Might, and D. V. Horn. Pushdown control-flow analysis for free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16. ACM, 2016.

[14] R. Glück. Simulation of two-way pushdown automata revisited. In A. Banerjee, O. Danvy, K.-G. Doh, and J. Hatcliff, editors, *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, Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2013.

[15] R. Hinze. Deriving backtracking monad transformers. In *ACM SIGPLAN Notices*, volume 35, pages 186–197. ACM, 2000.

[16] G. Janssens and K. F. Sagonas. On the use of tabling for abstract interpretation: An experiment with abstract equation systems. In *TAPD*, pages 118–126. Citeseer, 1998.

[17] M. J. Jaskelioff. *Lifting of operations in modular monadic semantics*. PhD thesis, University of Nottingham, 2009.

[18] J. I. Johnson and D. Van Horn. Abstracting abstract control. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*. ACM, 2014.

[19] J. I. Johnson, I. Sergey, C. Earl, M. Might, and D. V. Horn. Pushdown flow analysis with abstract garbage collection. *Journal of Functional Programming*, 2014.

[20] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 1976.

[21] O. Kiselyov. Typed tagless final interpreters. In *Generic and Indexed Programming*, pages 130–174. Springer, 2012.

[22] O. Kiselyov, C. chieh Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers:(functional pearl). *ACM SIGPLAN Notices*, 40(9):192–203, 2005.

[23] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95. ACM, 1995.

[24] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343. ACM, 1995.

[25] M. Might. Logic-flow analysis of higher-order programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07. ACM, 2007.

[26] E. Moggi. An abstract view of programming languages. Technical report, Edinburgh University, 1989.

[27] P. C. Nguyễn and D. V. Horn. Relatively complete counterexamples for higher-order programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015. ACM, 2015.

[28] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[29] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95. ACM, 1995.

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

[31] I. Sergey, D. Devriese, M. Might, J. Midtgaard, D. Darais, D. Clarke, and F. Piessens. Monadic abstract interpreters. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2013.

[32] G. L. Steele,Jr. Building interpreters by composing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94. ACM, 1994.

[33] T. Swift and D. S. Warren. Xsb: Extending prolog with tabled logic programming. *Theory and Practice of Logic Programming*, 12(1-2): 157–187, 2012.

[34] H. Tamaki and T. Sato. Old resolution with tabulation. In *International Conference on Logic Programming*, pages 84–98. Springer, 1986.

[35] S. Tobin-Hochstadt and D. Van Horn. Higher-order symbolic execution via contracts. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA. ACM, 2012.

[36] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proceedings of the 32nd*

ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI. ACM, 2011.

[37] A. van der Ploeg and O. Kiselyov. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In *ACM SIGPLAN Notices*, volume 49, pages 133–144. ACM, 2014.

[38] D. Van Horn and M. Might. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP. ACM, 2010.

[39] D. Van Horn and M. Might. Systematic abstraction of abstract machines. *Journal of Functional Programming*, 2012.

[40] A. Vandenbroucke, T. Schrijvers, and F. Piessens. Fixing non-determinism. In *IFL 2015: Symposium on the implementation and application of functional programming languages Proceedings*, number 27. Association for Computing Machinery, 2016.

[41] D. Vardoulakis. *CFA2: Pushdown Flow Analysis for Higher-Order Languages*. PhD thesis, Northeastern University, 2012.

[42] D. Vardoulakis and O. Shivers. CFA2: a Context-Free approach to Control-Flow analysis. *Logical Methods in Computer Science*, 2011.