Abstracting Definitional Interpreters

Functional Pearl

Abstract

A definitional interpreter written in monadic style can express a wide variety of abstract interpretations. We give a rational reconstruction of a definitional abstract interpreter for a higher-order language by constructing a series of units implementing monadic operations. The denouement of our story is a sound and computable abstract interpreter that arises from the composition of simple, independent components. Remarkably, this interpreter implements a form of pushdown control flow analysis (PDCFA) 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.

Keywords definitional interpreters, abstract interpretation, pushdown control flow analysis

1. Introduction

In his landmark paper, *Definitional interpreters for higher-order languages* [13], Reynolds first observed that when a language is defined by way of an interpreter, it is possible for the defined language to inherit semantic characteristics of the defining language of the interpreter. For example, it is easy to write a compositional evaluator that defines a call-by-value language if the defining language is call-by-value, but defines a call-by-name language if the defining language is call-by-name.

In this paper, we make the following two observations:

1. Definitional interpreters, written in monadic style, can simultaneously define a language's semantics as well as safe approximations of those semantics.
2. These definitional abstract interpreters can inherit characteristics of the defining language. In particular, we show that the abstract interpreter inherits the call and return matching property of the defining language and therefore realizes an abstract intpretation in the pushown style of analyses [3, 19].

A common problem of past approaches to the control flow analysis of functional languages is the inability to properly match a function call with its return in the abstract semantics, leading to infeasible program (abstract) executions in which a return is made from one point in the program text, but control units to another. The CFA2 analysis of Vardoulakis and Shivers [20] was the first approach that overcame this shortcoming. In essence, this kind of analysis can be viewed as replacing the traditional finite automata abstractions of programs with pushdown automata [3].

In this paper we investigate the use of definitional interpreters as the basis for abstract interpretation of higher-order languages. We show that a definitional interpreter—a compositional evaluation function—written in a monadic and componential style can express a wide variety of concrete and abstract interpretations.

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 [18] is a recipe for transforming a machine semantics into an easily abstractable form. There are a few essential elements to the transformation:

- continuations are heap-allocated
- variable bindings are heap-allocated
- the range of the heap is changed from values to sets of values
- heap update is interpreted as a join
- heap dereference is interpreted as a non-deterministic choice

These transformations are semantics-preserving as the original and derived machines operate in a lock-step correspondence. But the real value of the derived semantics stems from the fact that it's possible to turn the derived machine into an abstract interpreter with two simple steps:

- bounding heap allocation to a finite set of addresses
- widening base values to some abstract domain

Moreover, the soundness of the resulting abstraction is self-evident and easily proved.

The AAM approach has been used to build static analyzers for languages such as Java, JavaScript, Racket, Coq, and Erlang. It's been applied to malware detection, contract verification, and ?.

Given the success of the AAM approach, it's natural to wonder what is essential about the low-level machine basis of the semantics and whether a similar approach is possible with a higher-level formulation of the semantics such as a compositional evaluation function.

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

Benefits of a definitional interpreter approach:

- As we will see, the definitional interpreter approach is not formulated as a transformation on the semantics itself, but rather uses alternative notions of a monad to express the "abstracting" transformations. This means the concrete and abstract interpreters for a language can share large parts of their implementation; there is just one interpreter with a multiplicity of interpretations.
- There is a rich body of work and many tools and techniques for constructing *extensible* interpreters, all of which applies to high-level semantics, not machines. By putting abstract interpretation

```
E  ::= (vbl X)        ; Variable
       (num Number)   ; Number
       (lam X E)      ; Lambda
       (ifz E E E)    ; Conditional
       (op1 O1 E)     ; Unary primitive
       (op2 O2 E E)   ; Binary primitive
       (app E E)      ; Application
       (lrc X E E)    ; Letrec
X  ::= Symbol         ; Variable name
O1 ::= add1 ...       ; Unary operator
O2 ::= + - ...        ; Binary operator
```

Figure 1: Syntax

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 particular, we use *monad transformers* to build re-usable components for mixing and matching the constiuent parts of an abstract interpreter.

- Finally, using definitional interpreters for abstract interpretation satisfies an intellectual itch that asks whether it can be done at all. In solving this technical challenge, we discover a pleasant surprise about the definitional interpreter approach: it is inherently "pushdown," a property that has been the subject of several papers. Under the interpreter approach, it comes for free.

3. A Definitional Interpreter

We begin by first 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, unary and 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 functions with a closure data structure that pairs together the code with the environment in which a function definition was evaluated,
- it is structured monadically and uses monad operations to interact with the environment and heap,
- 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 x = e . r) ≡ (let ((x e)) (do . r))
(do e . r) ≡ (bind e (λ (_) (do . r)))
(do b) ≡ b
```

Finally, there are two unconvential 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` which it calls in order to recur. (We have employed a

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

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)))]
    [(op1 o e₀)
     (do v ← (ev e₀)
         (δ o v))]
    [(op2 o e₀ e₁)
     (do v₀ ← (ev e₀)
         v₁ ← (ev e₁)
         (δ o v₀ v₁))]
    [(lrc 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: Definitional interpreter

bit of cuteness by naming the first parameter `ev`, thereby shadowing the outer `ev` and making subsequent calls look like recursive calls.) 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. As we will see, this open recursive form will be crucial for interposition to collect information about the intensional properties of evaluation.

Second, the code is clearly *incomplete*. There are a number of free variables, noted in italics. These free variables fall into a few roles:

- providing the underlying monad of the interpreter: *return* and *bind*,
- providing an interpretation of primitives: *δ* and *zero?*,
- providing environment operations: *ask-env* for retreiving the environment and *local-env* for installing an environment,
- providing store operations: *ext* for updating the store, and *find* for dereferencing locations, and
- a remaining operation for *alloc*ating store locations, used to bind variables.

Going forward, we make frequent use of sets of definitions involving free variables, so we call such a collection a *component*.

monad@

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

δ@

```
(define (δ . ovs)
  (match ovs
    [(list 'add1 n)  (return (add1 n))]
    [(list 'sub1 n)  (return (sub1 n))]
    [(list '- n)     (return (- n))]
    [(list '+ n₀ n₁) (return (+ n₀ n₁))]
    [(list '- n₀ n₁) (return (- n₀ n₁))]
    [(list '* n₀ n₁) (return (* n₀ n₁))]
    [(list 'quotient n₀ n₁)
     (if (= 0 n₁)
         fail
         (return (quotient 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 interpreter

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]

Let us now examine a set of components for completing the definitional interpreter. figure 3 gives the definition for a series of components that complete the interpreter. The first and most magical component is `monad@`, which uses our `define-monad` macro to generate a set of bindings based on a monad transformer stack. For this interpreter, 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`.

We also add a `mrun` operation for running computations, which kicks off the computation by providing 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]))
```

And the remaining operations are straightforward, too. So the use of monad transformers can be seen as a mere convenience, but as we will see moving to more and more involved monad stacks, it's a useful one.

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.

The `store@` component defines the derived operations on stores of `find` and `ext` for finding and extending the store in terms of the monadic operations.

The only remaing 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 [16], we can easily 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 an answer consisting of a closure over the empty environment together 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:

```
> (quotient 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:

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

failures would not include the store:

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

At this point, we've defined a fairly run of the mill definitional interpreter. Despite these pedestrian beginnings, we essentially have the complete skeleton for everything to come. In particular, we will reuse `ev@` in all of the remaining interpreters. Now let's do something a bit more enchanting.

4. 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

[2] We use Racket *units* [4] to model components in our implementation.

trace-monad@

```
(define-monad
  (ReaderT (FailT (StateT
    (WriterT List ID)))))
```

ev-tell@

```
(define (((ev-tell ev₀) ev) e)
  (do ρ ← ask-env
      σ ← get-store
      (tell (list e ρ σ))
      ((ev₀ ev) e)))
```

Figure 4: Tracing variant

for a tracing interpreter and a kind of "mix-in" for the evaluator. The monad stack adds `WriterT` using `List`, which provides a new operation named `tell` for writing items to the stream of reached states. The `ev-trace` function is a wrapper around an underlying `ev` function which interposes itself between each recursive call by telling the current state of the evaluator, that is 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 the resulting answer and a list of all the states seen by the evaluator, in the order in which they were seen. For example:

```
> (* (+ 3 4) 9)
'((63 . ())
  ((* (+ 3 4) 9) () ())
  ((+ 3 4) () ())
  (3 () ())
  (4 () ())
  (9 () ()))
> ((λ (x) (λ (y) x)) 4)
'((((λ (y) x) . ((x . 0))) . ((0 . 4)))
  (((λ (x) (λ (y) x)) 4) () ())
  ((λ (x) (λ (y) x)) () ())
  (4 () ())
  ((λ (y) x) ((x . 0)) ((0 . 4))))
```

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.

So our setup makes it easy not only to express the run of the mill interpreter, but also different forms of collecting semantics. Let us now start to look at abstractions.

5. Abstracting Base Values

One of things an abstract interpreter must do in order to become decidable is to have some form of abstraction for the base types of the language. A very simple approach is to use a finite-element abstract domain. We can do this for our sole base type of numbers by introducing a new kind of number, written `'N`, which is an abstract value that stands for all numbers. Abstract values will be introduced by alternative interpretation of the primitive operations, given in figure 5, which simply produces `'N` in all cases. Some care must be taken in the interpretation of `'quotient` since if the denominator is an abstract value, the result must include a failure since `0` is in the set of values abstracted by `'N`. This means that dividing a number

δ^@

```
(define (δ . ovs)
  (match ovs
    [(list 'add1 n)  (return 'N)]
    [(list 'sub1 n)  (return 'N)]
    [(list '+ n₀ n₁) (return 'N)]
    [(list '- n₀ n₁) (return 'N)]
    [(list '* n₀ n₁) (return 'N)]
    [(list 'quotient n₀ (? number? n₁))
     (if (= 0 n1) fail (return 'N))]
    [(list 'quotient n₀ n₁)
     (mplus (return 'N) fail)]))

(define (zero? v)
  (match v
    ['N (mplus (return #t) (return #f))]
    [_  (return (= 0 v))]))
```

Figure 5: Abstracting primitive operations

by an abstract value must produce *two answers*: `'N` and `'failure`. This is done by adding non-determinism to the monad stack,

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

which provides a `mplus` operation for combining multiple answers. Non-determinism is also used in the implementation of `zero?`, which returns both true and false on `'N`.

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

```
> (* (+ 3 4) 9)
(set '(N . ()))
> (quotient 5 (add1 2))
(set '(failure . ()) '(N . ()))
> (if0 (add1 0) 3 4)
(set '(3 . ()) '(4 . ()))
```

If we were to link together the abstract variant of `δ` with the *tracing* monad stack with non-determinism added in,

```
(ReaderT (FailT (StateT
  (WriterT List (NondetT ID)))))
```

we would get an evaluator that produces sets of traces:

```
> (if0 (add1 0) 3 4)
(set
 '((3 . ())
   ((if0 (add1 0) 3 4) () ())
   ((add1 0) () ())
   (0 () ())
   (3 () ()))
 '((4 . ())
   ((if0 (add1 0) 3 4) () ())
   ((add1 0) () ())
   (0 () ())
   (4 () ())))
```

It should be clear that the interepreter will only ever see a finite set of numbers (including `'N`), but it's definitly not true that the interpreter halts on all inputs. Firstly, it's still possible to generate an infinite number of closures. Secondly, there's no way for the interpreter to detect when it sees a loop. To make a terminating abstract

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

                                                    store-nd@
(define (find a)
  (do σ ← get-store
      (for/monad+ ([v (σ a)])
        (return v))))

(define (ext a v)
  (update-store
    (λ (σ) (σ a (if (∈ a σ)
                    (set-add (σ a) v)
                    (set v))))))
```

Figure 6: Abstracting allocation: 0CFA

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

Figure 7: Caching, first attempt

interpreter requires tackling both. Let's look next at abstracting closures.

6. 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 abstract closures by abstracting the set of addresses. Following the AAM approach, we can do this by modifying the allocation function to always produce elements drawn from a finite set. In order to retain soundness in the semantics, we will need to modify the store to map addresses to *sets* of values and model store update as a join and 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 variable bindings by using a variable's name as its address. This gives a monomorphic, or 0CFA-like, abstraction.

Figure 6 shows an alternative component for finite allocation that uses variables names as the notion of addresses and a component for the derived operations `find` and `ext` when the store uses a *set* as its range. The `for/monad+` form is just a convenience for combining a set of computations with `mplus`; in other words, `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 together with the same monad stack from Section 5, we obtain an interpreter that loses precision whenever variables are bound to multiple values. For example, this program binds `x` to both `0` and `1` and therefore produces both answers when run:

```
> (let f (λ (x) x)
    (let _ (f 0) (f 1)))
(set
 '(0 . ((f ((λ (x) x) . ())) (x 1 0)))
 '(1 . ((f ((λ (x) x) . ())) (x 1 0))))
```

We've now taken care of making a sound, finite abstraction of the space of all closures that arise during evaluation. It would seem we are very close to having a sound, total abstract interpretation function.

7. Detecting Cycles with a Cache

At this point, it's worth observing that the interpreter obtained by linking together `δ^@` and `alloc^@` components will only ever visit a finite number of states for a given program. The state consists of an expression, environment, and store mapping addresses to sets of values. To see that this is a finite set is pretty straightforward: expressions (in the given program) are finite, environments are maps from variables (again, finite in a program) to address. The addresses are finite thanks to `alloc^`, so environments are finite. The store maps addresses (finite by `alloc^`) to sets of values. The values are base values, which are finite by `δ^`, or closures which consist of expressions (in the given program) and an environment, which we just seen are finite. Since the elements of the sets are finite, the sets themselves are finite, and therefore stores, and finally states, are all finite sets.

The problem is that while the interpreter only ever see a finite set of inputs, it *doesn't know it*. So even a simple loop will cause the interpreter to diverge:

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

To solve this problem, let's introduce a notion of a *cache* which is a mapping from states to sets of values. The basic idea is that we will use the cache to do a form of co-inductive programming. While evaluating a state `ς` for the first time, we may at some point be asked to evaluate exactly `ς` again. Should this happen, we can return the empty set of results. We will use the cache to track the encountered states and the results they produce. By maintaining the cache, we avoid the possibility of diverging.

We use the following monad stack, which adds a "cache" component, which will be a finite map from states to sets of values:

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

The `CacheT` monad transformer is a slight variation of `StateT`, with operations `get-$` and `update-$` for getting and updating the cache, respectively. The difference between this and the state monad is it that joins finite maps by union of the range in its `mplus` operation.

Figure 7 gives an `ev`-wrapper that interposes itself on each recursive call to do the following steps:

Check if the current state is in the cache. If it's in the cache, return all the results given in the cache. If it's not, set the cache for the current state to the empty set, evaluate the expression, add the resulting value to the cache for the state, and return the result.

We can now define an evaluation function that mixes in `ev-cache`:

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

If we were to link this together with `alloc@` and `δ@`, we'd obtain a concrete interpreter that either 1) produces the empty set because it encountered a loop, 2) produces a singleton result, or 3) diverges because it encounters an infinite set of states. But if we were to link this together with `alloc^@` and `δ^@`, we'd obtain an abstract interpreter that is *total*: it terminates on all inputs.

To see why this, observe that for a given program there only a finite set of possible caches. We have already seen that there are a finite set of states and values, so it follows that there are only a finite set of maps from states to sets of values. Now notice that on each recursive call, either the state is in the cache and it returns immediately, or the cache grows. So programs simply cannot run forever because that would imply the cache would grow forever.

It should be easy to see that if evaluating a state `ς` requires recursively evaluating that same state, it will now produce the empty set since the cache will be updated to map `ς` to `⌀` before proceeding to the sub-expressions.

We can now see that the caching abstract interpreter halts on programs that loop (for simplicity, the cache and store are omitted from the printed results):

```
> (rec f (λ (x) (f x)) (f 0))
(set)
```

This accomplishes the goal of terminating on this example, and it even gives the right answer—the empty set—since this program produces no results.

It also works for recursive functions that terminate in the concrete, but have loops once abstracted:

```
> (rec fact (λ (n)
              (if0 n 1 (* n (fact (sub1 n)))))
    (fact 5))
(set 'N)
```

It may seem we've accomplished our goal of making a sound and decidable abstract interpreter. However this approach is broken in general: it is not sound in the presence of abstraction. The problem here is that when the interpreter reaches "the same" state it has seen before, what we mean by "the same" in the presence of abstraction is subtle. For example, imagine evaluating a function application of some function `f` to an abstract value `'N`. Suppose in evaluating this application we encounter another application of `f` to `'N`. Is it the same application? Well, yes and no. It is the same *abstract* state, however the abstract state stands for a set of concrete states; in this case, the application of `f` to all numbers. So there are states stood for in the abstraction that are equal *and* not equal. In other words, in the presence of abstraction, when a loop is detected, there *may* be a loop in the concrete interpretation. Our naive loop detection set-up however is assuming there *must* be a loop.

We can demonstrate the problem with a simple counter-example to soundness:

```
> (rec f (λ (x)
           (if0 x 0 (if0 (f (sub1 x)) 2 3)))
    (f (add1 0)))
(set 0)
```

`ev-cache@`

```
(define (((ev-cache ev₀) ev) e)
  (do ρ ← ask-env
      σ ← get-store
      ς = (list e ρ σ)
      Σ ← get-$
      (if (∈ ς Σ)
          (for/monad+ ([v×σ (Σ ς)])
            (do (put-store (cdr v×σ))
                (return (car v×σ))))
          (do Σ⊥ ← ask-⊥
              ; initialize to prior, if exists
              (put-$ (Σ ς (if (∈ ς Σ⊥) (Σ⊥ ς) ⌀)))
              v  ← ((ev₀ ev) e)
              (update-$
                (λ (Σ)
                  (Σ ς (set-add (Σ ς)
                                (cons v σ)))))
              (return v)))))
```

Figure 8: Caching, with fall-back to prior

Concretely, this program always returns `2`, however with the combination of loop detection and abstraction determines that this program always produces `0`. That's clearly unsound.

8. Fixing the Cache

The basic problem with the caching solution of section 7 is that it cuts short the exploration of the program's behavior. In the soundness counter-example, the inner call to `f` is present in the cache so neither branch of the conditional is taken; it is at this point of bottoming out that we determine `f` may return `0`. Of course, now we know that the conditional should have take the true branch since `0` could be returned, but it's too late: the program has terminated.

To restore soundness, what we need to do is somehow *iterate* the interpreter so that we can re-explore the behavior knowing that `f` may produce `0`. A first thought may be to do a complete evaluation of the program, take the resulting cache, and then feed that in as the initial cache for a re-evaluation of the program. But there's an obvious problem... doing so would result in a cache hit and the saved results would be returned immediately without exploring any new behavior.

The real solution is that we want to use the prior evaluation's cache as a kind of co-inductive hypothesis: it's only when we detect a loop that we want to produce all of the results stored in the prior cache. This suggests a two-cache system in which the prior cache is only used when initializing the local cache. In other words, we want to use the prior cache entry in the place of `⌀`. When iterating the interpreter, we always start from an empty local cache and fall back on the prior cache results when initializing the cache entry before making a recursive call. Since the prior cache is never written to, we can model the prior cache as a reader monad and add it to the stack:

```
(ReaderT (FailT (StateT (NondetT
  (ReaderT ; the prior cache
    (CacheT ID))))))
```

The revised `ev-cache@` component is given in figure 8, which uses the `ask-⊥` operation to retreive the prior cache. If the prior cache is empty, this code degenerates into exactly what was given in figure 7.

fix-cache@

```
(define ((fix-cache eval) e)
  (do ρ ← ask-env
      σ ← get-store
      ς ≔ (list e ρ σ)
      (mlfp (λ (Σ) (do (put-$ ⌀-map)
                       (put-store σ)
                       (local-⊥ Σ (eval e))
                       get-$)))
      Σ ← get-$
      (for/monad+ ([v×σ (Σ ς)])
        (do (put-store (cdr v×σ))
            (return (car v×σ))))))

(define (mlfp f)
  (let loop ([x ⌀-map])
    (do x′ ← (f x)
        (if (equal? x′ x)
            (return (void))
            (loop x′)))))
```

Figure 9: Finding fixed-points in the cache

We are left with two remaining problems; we need to figure out: 1) how to pipe the cache from one run of the interpreter into the next and 2) when to stop. The answer to both is given in figure 9.

The `fix-cache` function takes a *closed evaluator*, i.e. something of the form `(fix ev)`. It iteratively runs the evaluator. Each run of the evaluator resets the "local" cache to empty and uses the cache of the previous run as it's fallback cache (initially it's empty). The computation stops when a least fixed-point in the cache has been reached, that is, when running the evaluator with a prior gives no changes in the resulting cache. At that point, the result is returned.

With these peices in place, we can construct an interpreter as:

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

When linked with `δ^` and `alloc^`, this interpreter is a sound, computable abstraction of the original definitional interpreter. Note that the iterated evaluator always terminates: the cache resulting from each run of the evaluator contains *at least* as much information as the prior cache, each run of the evaluator terminates, so the iterated evaluator terminates by the same principle as before: the cache monotonically grows and is finite in size.

We have thus achieved our goal and can confirm it gives the expected answers on the previous examples:

```
> (rec f (λ (x) (f x)) (f 0))
(set)
> (rec fact (λ (n)
              (if0 n 1 (* n (fact (sub1 n))))))
    (fact 5))
(set 'N)
> (rec f (λ (x)
           (if0 x 0 (if0 (f (sub1 x)) 2 3)))
     (f (add1 0)))
(set 3 0 2)
```

Let us now take stock of what we've got.

9. 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 are we really doing?

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: continuations are not store-allocated.

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 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.

10. Symbolic Execution and Path-Sensitive Verification

We first present an extension to the monad stack and metafunctions that gives rise to a symbolic execution [9], then show how abstractions discussed in previous sections can be applied to enforce termination, turning a traditional symbolic execution into a path-sensitive verification.

10.1 Symbolic Execution

Figure 10 shows the units needed to turn the existing interpreter into a symbolic executor, in addition to adding symbolic numbers `(sym X)` to the language syntax. Primitives such as `'quotient` now may 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 known to have evaluated to `0`. This set is another state component provided by `StateT`. Monadic operations *`get-path-cond`* and *`refine`* reference and update the path-condition. 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 besides refining the path-condition with the assumption made. Operator `'¬` represents negation in our language.

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

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

A scaled up symbolic executor can have *`zero?`* calling out to an SMT solver for interesting arithmetics, and extend the language with symbolic functions and blame semantics for sound higher-order symbolic execution [12, 17].

10.2 From Symbolic Execution to Verification

Traditional symbolic executors mainly aim to find bugs and provide no termination guarantee. We can apply abstracting units presented in previous sections, namely base value widening (section 5), finite allocation (section 6), caching and fixing (section 7 and section 8) to turn a symbolic execution into a sound, path-sensitive program verification.

```
E  ::= ... (sym X)  ; Symbolic number
```

symbolic-monad@

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

ev-symbolic@

```
(define (((ev-symbolic ev₀) ev) e)
  (match e
    [(sym x) (return x)]
    [e       ((ev₀ ev) e)]))
```

δ-symbolic@

```
(define (δ . ovs)
  (match ovs
    ...
    [(list 'quotient v₀ v₁)
     (do z? ← (zero? v₁)
         (cond
          [z? fail]
          [(and (number? v₀) (number? v₁))
           (return (quotient v₀ v₁))]
          [else
           (return `(quotient ,v₀ ,v₁))]))]
    [(list '¬ 0) 1]
    ...))

(define (zero? v)
  (do φ ← get-path-cond
      (match v
        [(? number? n) (return (= 0 n))]
        [v #:when (∈ v φ) (return #t)]
        [v #:when (∈ v `(¬ ,v)) (return #f)]
        [`(¬ ,v′) (do a ← (zero? v′)
                      (not a))]
        [v (mplus (do (refine v)
                      (return #t))
                  (do (refine `(¬ ,v))
                      (return #f)))])))
```

Figure 10: Symbolic execution variant

Operations on symbolic values introduce a new source of infinite configurations by building up new symbolic values. We therefore straightforwardly widen a symbolic value to the abstract number `'N` when it shares an address with a different number. Figure 11 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`.

11. Try It Out

All of the components discussed in this paper have been implemented as units [4] in Racket [5]. We have also implemented a `#lang` langauge 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):

```
raco pkg install \
```

δ-symbolic@

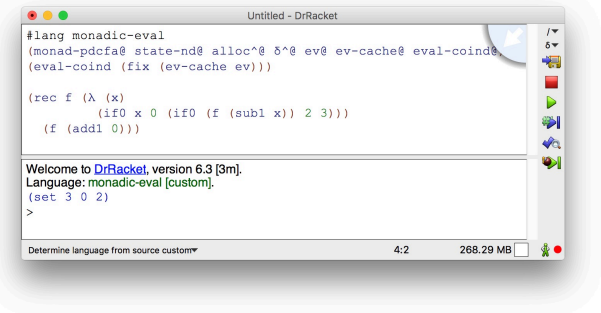
```
(define (δ . ovs)
  (match ovs
    ...
    [(list 'quotient v₀ v₁)
     (do z? ← (zero? v₁)
         (cond
          [z? fail]
          [else
           (match (list v₀ v₁)
            [(list (? number? n₀) (? number? n₁))
             (return (quotient n₀ n₁))]
            [(list _ ... 'N _ ...)
             (return 'N)]
            [(list v₀ v₁)
             (return `(quotient ,v₀ ,v₁))])]))]
    ...))

(define (zero? v)
  (do φ ← get-path-cond
      (match v
        ['N (mplus (return #t) (return #f))]
        ...)))
```

Figure 11: Symbolic execution with abstract numbers

```
https://github.com/<anon>/monadic-eval.git
```

A `#lang monadic-eval` program starts with a list of components, which are linked together, and an expression producing an evaluator. Subsequent forms are interpreted as expressions when run. Programs can be run from the command-line or interactively in the DrRacket IDE. For example, here is a screen shot of the PDCFA evaluator running the example from section 8.



12. Related Work

Danvy, monadic interpreters and abstract machines. [1]

12.1 Pushdown

[6]

12.2 Monadic interpreters

[10, 15]

12.3 Monadic abstract interpreters

PLDI 2013: small-step monad. [14].

[2]

[8]

12.4 Big CFA2

[19]
[7]
HOFA and SE:
[11]

Bibliography

[1] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science* 342(1), 2005.

[2] David Darais, Matthew Might, and David Van Horn. Galois Transformers and Modular Abstract Interpreters. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

[3] Christopher Earl, Matthew Might, and David Van Horn. Pushdown Control-Flow Analysis of Higher-Order Programs. In *Proc. Workshop on Scheme and Functional Programming*, 2010.

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

[5] Matthew Flatt and PLT. Reference: Racket. PLT Design Inc., PLT-TR-2010-1, 2010.

[6] Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown Control-Flow Analysis for Free. In *Proc. 43rd ACM SIGPLAN-SIGACT Symposium on Principles in Programming Languages*, 2016.

[7] Robert Glück. Simulation of two-way pushdown automata revisited. *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday*, 2013.

[8] J. Ian Johnson and David Van Horn. Abstracting Abstract Control. In *Proc. 10th ACM Symposium on Dynamic Languages*, 2014.

[9] James C. King. Symbolic Execution and Program Testing. In *Proc. Communications of the ACM*, 1976.

[10] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proc. 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995.

[11] Matthew Might. Logic Flow Analysis of Higher-Order Programs. In *Proc. 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.

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

[13] John Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. ACM Annual Conference*, 1972.

[14] Ilya Sergey, Dominique Divriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. Monadic Abstract Interpreters. In *Proc. ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2013.

[15] Guy L. Steele Jr. Building interpreters by composing monads. In *Proc. 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1994.

[16] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as Libraries. In *Proc. ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2011.

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

[18] David Van Horn and Matthew Might. Abstracting Abstract Machines. In *Proc. ACM International Conference on Functional Programming*, 2010.

[19] Dimitris Vardoulakis. CFA2: Pushdown Flow Analysis for Higher-Order Languages. PhD dissertation, Northeastern University, 2012.

[20] Dimitris Vardoulakis and Olin Shivers. CFA2: a Context-Free Approach to Control-Flow Analysis. *Logical Methods in Computer Science* 7(2:3), 2011.