Monadic Abstract Interpreters

Ilya Sergey
IMDEA Software Institute, Spain
ilya.sergey@imdea.org

Dominique Devriese
iMinds – DistriNet, KU Leuven, Belgium
dominique.devriese@cs.kuleuven.be

Matthew Might
University of Utah, USA
might@cs.utah.edu

Jan Midtgaard
Aarhus University, Denmark
jmi@cs.au.dk

David Darais
Harvard University, USA
darais@seas.harvard.edu

Dave Clarke Frank Piessens
iMinds – DistriNet, KU Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be

Abstract

Recent developments in the systematic construction of abstract interpreters hinted at the possibility of a broad unification of concepts in static analysis. We deliver that unification by showing context-sensitivity, polyvariance, flow-sensitivity, reachability-pruning, heap-cloning and cardinality-bounding to be independent of any particular semantics. Monads become the unifying agent between these concepts and between semantics. For instance, by plugging the same "context-insensitivity monad" into a monadically-parameterized semantics for Java or for the lambda calculus, it yields the expected context-insensitive analysis.

To achieve this unification, we develop a systematic method for transforming a concrete semantics into a monadically-parameterized abstract machine. Changing the monad changes the behavior of the machine. By changing the monad, we recover a spectrum of machines—from the original concrete semantics to a monovariant, flow- and context-insensitive static analysis with a singly-threaded heap and weak updates.

The monadic parameterization also suggests an abstraction over the ubiquitous monotone fixed-point computation found in static analysis. This abstraction makes it straightforward to instrument an analysis with high-level strategies for improving precision and performance, such as abstract garbage collection and widening.

While the paper itself runs the development for continuation-passing style, our generic implementation replays it for direct-style lambda-calculus and Featherweight Java to support generality.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis, Operational semantics

General Terms Languages, Theory

Keywords abstract machines, abstract interpretation, monads, operational semantics, collecting semantics, abstract garbage collection, interpreters

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

1. Introduction

Recent work on systematizing the construction of abstract interpreters [15, 23, 24] hints at the possibility of broad theoretical unification within static analysis. Van Horn and Might [23] sketch a method for abstracting abstract machines into static analyzers by bounding the store of the abstract machine: once the store is bounded, the abstraction and then the analysis follows.

But, bounding the store is an act of design—of human intervention. How a designer bounds the store immediately determines classical properties of the analysis such as its context-sensitivity and its polyvariance. While not directly expressed in terms of a bound on the store, other classical properties are also related to the abstraction and handling of the store, including heap-cloning, reachability-pruning and cardinality-bounding. But, other properties, such as flow-sensitivity, path-sensitvity and some kinds of widening, have little to do with the store, and more to with the (re-)interpretation of the abstract semantics during analysis.

Fortunately, there is a construct that encompasses all of these concerns: the monad. Monads were originally adapted to programming languages to provide a durable abstraction for mutation in a purely functional language. As such, expressing a semantics monadically allows the monad to fully veil the details of interacting with a store. Yet monads have always provided more than just a means for hiding effects in a purely functional manner: they also allow a near-complete reinterpretation of computations expressed monadically, *e.g.*, the instant and powerful non-determinism of the list monad. This semantic reflection in precisely the right dimensions allows monads to encapsulate a variety of concepts in static analysis.

The payoff of this realization is immediate: we can monadically refactor semantics for languages as diverse as the lambda calculus and Java, yet define notions like context-sensitivity for both at the same time, with the same monad.

Our presentation details every step of the monadic refactoring for continuation-passing style lambda calculus, and then develops the monadic parameters that induce static analyzers. The implementation of the approach in the accompanying code repository replays the same monadic refactoring for a direct-style lambda calculus and for Featherweight Java. *The monads remain the same.*

We have chosen Haskell in lieu of formal mathematics for the presentation for two reasons: (1) Haskell is directly executable, and (2) Haskell has concise, convenient and readable syntax for expressing monads—the central actor in our work. Our fundamental results are no more restricted to Haskell than monads are restricted to Haskell.

1.1 Overview

Van Horn and Might's systematic abstraction relies on breaking down recursive structure in the concrete state-space and threading that structure through a store. This refactoring of the state-space imposes a corresponding store-passing-style transformation on the semantics.

Were we writing an *interpreter in Haskell* rather than a *formal semantics*, we would recognize store-passing style as an anti-pattern: in Haskell, monads are the right generalization for capturing artifacts like a store in a purely functional manner. When we apply a monadic transformation in lieu of a store-passing transformation to a concrete semantics, the resulting monad erupts as the catalyst for unifying what appeared as *ad hoc* choices in the design of classical static analyses. Even the nondeterminism that arises during abstraction of an operational semantics can be captured, explained and throttled entirely monadically.

By delegating interaction with the store into a monad, the monad determines not only the polyvariance and context-sensitivity of the analysis, but also the degree to which the analysis prunes the abstract heap based on reachability (abstract garbage collection) and bounds the cardinality of abstract addresses for shape analysis. In fact, it suggests a refactoring of the traditional fixed-point iteration such that path-sensitivity and flow-sensitivity also fall out as natural parameters.

Our monadic abstraction of the semantics orthogonalizes the classic dimensions of static analysis, independent of the specific semantics in use.

We illustrate a systematic method for transforming a concrete semantics into a monadically-parameterized machine, comprising both *concrete* and *abstract* interpretation, such that the monad determines the classical properties of an analysis. As in recent work on abstracting abstract machines [23], our semantic transformation implicitly utilizes the techniques of Danvy *et al.* [1, 2, 7] in order to calculate an abstract[1] machine equivalent to the concrete semantics. It diverges with this line of work by applying a monadic-normal-form transformation [6] (instead of a store-passing-style transformation) to the rules for the machine.

1.2 Contributions

- The central theoretical contribution of the paper is identifying and employing monads as a mechanism to abstract over the fundamental characteristics of the analysis.
- The central practical contribution is an executable proof-of-concept implementation of the described decomposition for a series of calculi.[2]
- We decouple the interpretation of the semantics from a monotonic fixed-point computation, which makes it possible to define analysis widening strategies independently of the semantics and of the analysis' other parameters.
- We illustrate degrees of freedom when constructing the analysis using our framework and show that components implementing non-deterministic transitions, polyvariance and abstract counting are semantics-independent and can be reused for different calculi (*e.g.*, Java and the lambda calculus) and analysis families.

[1] *Abstract* in the sense that it models the salient intensional behavior of a program.

[2] The implementation is available for the lambda-calculus (both in the form of CPS [15] and CESK-machine [23]) and Featherweight Java [19]:

`http://github.com/ilyasergey/monadic-cfa`

$$
\begin{aligned}
v &\in \mathsf{Var} \text{ is a set of identifiers} \\
lam &\in \mathsf{Lam} ::= (\lambda\ (v_1 \ldots v_n)\ call) \\
f, \text{æ} &\in \mathsf{AExp} = \mathsf{Var} + \mathsf{Lam} \\
call &\in \mathsf{Call} ::= (f\ \text{æ}_1 \ldots \text{æ}_n) + \mathsf{Exit}
\end{aligned}
$$

$$
\begin{aligned}
\varsigma \in \Sigma &= \mathsf{Call} \times Env & \rho \in Env &= \mathsf{Var} \rightharpoonup D \\
d \in D &= Clo & clo \in Clo &= \mathsf{Lam} \times Env
\end{aligned}
$$

Figure 1: A grammar for CPS and a concrete state-space.

2. Setting the Scene: Analyzing Continuation-Passing Style

We start our discovery of the monadic refactoring process with a minimalist higher-order language: continuation-passing style λ-calculus (CPS). We will apply the systematic abstraction process as described by Van Horn and Might [23] in full to CPS. Afterward, we'll modify the process by transforming the semantics into monadic normal form after the store-passing transformation. Thus, all interaction with the store will pass through a monad. Because interaction with the store is central to describing facets of modern flow analysis, such as context-sensitivity and heap-cloning, we will be able to describe these facets independently of a particular semantics.

In CPS (Figure 1), the lambda calculus is partitioned into two worlds: call sites and atomic expressions. Atomic expressions are lambda terms and variable references. Call sites encode the application of a function to arguments. A classical abstract machine for CPS [9] needs only two components in its state-space Σ—see Figure 1—a control component (Call) and an environment (Env). The domain D contains denotable values. CPS is so simple that there is only one kind of denotable value—closures.

The injector $\mathcal{I} : \mathsf{Call} \to \Sigma$ maps a program into this state-space:

$$\mathcal{I}(call) = (call, []),$$

In CPS, there is only one rule to describe the transition relation, $(\Rightarrow) \subseteq \Sigma \times \Sigma$. We don't write index ranges explicitly for series of meta-expressions, assuming $i = 1..n$ is obvious form the context.

$$
\begin{aligned}
(\llbracket (f\ \text{æ}_1 \ldots \text{æ}_n) \rrbracket, \rho) &\Rightarrow (call, \rho''), \text{where} \\
(\llbracket (\lambda\ (v_1 \ldots v_n)\ call) \rrbracket, \rho') &= \mathcal{A}(f, \rho) \\
d_i = \mathcal{A}(\text{æ}_i, \rho) \qquad & \rho'' = \rho'[v_i \mapsto d_i],
\end{aligned}
$$

where the evaluator $\mathcal{A} : \mathsf{AExp} \times Env \rightharpoonup Clo$ evaluates an atomic expression:

$$\mathcal{A}(v, \rho) = \rho(v) \qquad \mathcal{A}(lam, \rho) = (lam, \rho).$$

2.1 Attempting structural abstraction

A structural abstraction carries abstraction component-wise across the state-space and then lifts naturally over internal domains. However, a structural abstraction of the state-space for CPS yields:

$$
\begin{aligned}
\hat{\Sigma} &= \mathsf{Call} \times \widehat{Env} & \widehat{Env} &= \mathsf{Var} \rightharpoonup \hat{D} \\
\hat{D} &= \mathcal{P}(\widehat{Clo}) & \widehat{Clo} &= \mathsf{Lam} \times \widehat{Env}.
\end{aligned}
$$

A structural abstraction preserves mutual recursion between closures and environments, and with it, the unboundedness of the abstract state-space. Since our goal with this abstraction was a finite abstract state-space (hence a trivially computable abstract semantics), structural abstraction alone is insufficient.

2.2 Cutting recursion with a store

To arrive at a finite state-space, the systematic method detailed in "Abstracting Abstract Machines" [23] calls for transforming the abstract machine into store-passing style, which cuts the recursive knot between environments and closures by introducing addresses. With the introduction of a store, $\sigma \in \mathit{Store} = \mathit{Addr} \rightharpoonup D$, the store-passing transform produces the following state-space:

$$\begin{aligned} \varsigma \in \Sigma &= \mathsf{Call} \times \mathit{Env} \times \mathit{Store} \\ \rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathit{Addr} \\ d \in D &= \mathit{Clo} \\ \mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{Env} \\ a \in \mathit{Addr} &\text{ is an infinite set of addresses.} \end{aligned}$$

The modification of the transition relation is straightforward:

$$\begin{aligned} &\overbrace{(\llbracket (f\ \text{æ}_1 \ldots \text{æ}_n) \rrbracket, \rho, \sigma)}^{\varsigma} \Rightarrow (\mathit{call}, \rho'', \sigma'), \text{ where} \\ &(\llbracket (\lambda\ (v_1 \ldots v_n)\ \mathit{call}) \rrbracket, \rho') = \mathcal{A}(f, \rho, \sigma) \\ &\quad d_i = \mathcal{A}(\text{æ}_i, \rho, \sigma) \qquad a_i = \mathit{alloc}(v_i, \varsigma) \\ &\quad \rho'' = \rho'[v_i \mapsto a_i] \qquad \sigma' = \sigma[a_i \mapsto d_i]. \end{aligned}$$

The evaluator $\mathcal{A} : \mathsf{AExp} \times \mathit{Env} \times \mathit{Store} \rightharpoonup D$ is modified to take the store as an additional argument:

$$\mathcal{A}(v, \rho, \sigma) = \sigma(\rho(v)) \qquad \mathcal{A}(\mathit{lam}, \rho, \sigma) = (\mathit{lam}, \rho).$$

And, the (presently opaque) address-allocator $\mathit{alloc} : \mathsf{Var} \times \Sigma \to \mathit{Addr}$ yields a fresh address for each variable.

2.3 A second attempt at abstraction

With the recursion sliced from the state-space by the store-passing transformation, a structural abstraction succeeds in producing a finite abstract state-space:

$$\begin{aligned} \hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Call} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \\ \hat{\rho} \in \widehat{\mathit{Env}} &= \mathsf{Var} \rightharpoonup \widehat{\mathit{Addr}} \\ \hat{\sigma} \in \widehat{\mathit{Store}} &= \widehat{\mathit{Addr}} \to \mathcal{P}(\hat{D}) \\ \hat{d} \in \hat{D} &= \widehat{\mathit{Clo}} \\ \widehat{\mathit{clo}} \in \widehat{\mathit{Clo}} &= \mathsf{Lam} \times \widehat{\mathit{Env}} \\ \hat{a} \in \widehat{\mathit{Addr}} &\text{ is a finite set of abstract addresses,} \end{aligned}$$

and it induces a straightforward abstract transition relation:

$$\begin{aligned} &\overbrace{(\llbracket (f\ \text{æ}_1 \ldots \text{æ}_n) \rrbracket, \hat{\rho}, \hat{\sigma})}^{\hat{\varsigma}} \rightsquigarrow (\mathit{call}, \hat{\rho}'', \hat{\sigma}'), \text{ if} \\ &(\llbracket (\lambda\ (v_1 \ldots v_n)\ \mathit{call}) \rrbracket, \hat{\rho}') \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \\ &\quad \hat{d}_i \in \hat{\mathcal{A}}(\text{æ}_i, \hat{\rho}, \hat{\sigma}) \qquad \hat{a}_i = \widehat{\mathit{alloc}}(v_i, \hat{\varsigma}) \\ &\quad \hat{\rho}'' = \hat{\rho}'[v_i \mapsto \hat{a}_i] \qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \{\hat{d}_i\}]. \end{aligned}$$

Branching to every possible abstract closure introduces a subtle nondeterminism.

Naturally, the abstract argument evaluator wraps closures as singletons, but looks up variables as in the concrete version:

$$\hat{\mathcal{A}}(v, \hat{\rho}, \hat{\sigma}) = \hat{\sigma}(\hat{\rho}(v)) \qquad \hat{\mathcal{A}}(\mathit{lam}, \hat{\rho}, \hat{\sigma}) = \{(\mathit{lam}, \hat{\rho})\}.$$

The join on the abstract store allows each address in the finite set of abstract addresses to soundly represent multiple concrete addresses:

$$\hat{\sigma} \sqcup \hat{\sigma}' = \lambda \hat{a}.\hat{\sigma}(\hat{a}) \cup \hat{\sigma}'(\hat{a}).$$

And, it should now be clear that the abstract allocator $\widehat{\mathit{alloc}} : \mathsf{Var} \times \hat{\Sigma} \to \widehat{\mathit{Addr}}$ determines the polyvariance of the analysis (*i.e.*, distinguishing between bindings of the same variable in different evaluation contexts [22, 19]) because $\widehat{\mathit{alloc}}$ determines how many abstract variants are associated with each variable. For instance, allocating each time a new address corresponds to associating just *one* value with a variable at each moment of the program execution.

2.3.1 Example: Monovariant analysis (0CFA)

For example, a classical monovariant analysis (0CFA) comes from defining the set of abstract addresses to be the set of variables ($\widehat{\mathit{Addr}}_{\text{0CFA}} = \mathsf{Var}$), and then using variables as their own abstract addresses:

$$\widehat{\mathit{alloc}}_{\text{0CFA}}(v, \hat{\varsigma}) = v.$$

2.4 Introducing context with time-stamps

Van Horn and Might [23] realize that the abstract state-space as it stands lacks sufficient information to instantiate classical strategies for polyvariance (like k-CFA). To fix this, they introduce time-stamps (to the concrete and abstract semantics) as a new component of the state.

Time-stamps are used to remember execution context, and are directly responsible for the context-sensitivity of the analysis:

$$\begin{aligned} \hat{\varsigma} \in \hat{\Sigma} &= \mathit{Call} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{Time}} \\ \hat{t} \in \widehat{\mathit{Time}} &\text{ is a finite set of abstract times.} \end{aligned}$$

Each transition augments the time through an opaque function, $\widehat{\mathit{tick}} : \widehat{\mathit{Clo}} \times \hat{\Sigma} \to \widehat{\mathit{Time}}$ and by giving the allocator $\widehat{\mathit{alloc}} : \mathsf{Var} \times \widehat{\mathit{Time}} \to \widehat{\mathit{Addr}}$ access to this context instead of the whole state:

$$\begin{aligned} &\overbrace{(\llbracket (f\ \text{æ}_1 \ldots \text{æ}_n) \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{t})}^{\hat{\varsigma}} \rightsquigarrow (\mathit{call}, \hat{\rho}'', \hat{\sigma}', \hat{t}'), \text{ if} \\ &\underbrace{(\llbracket (\lambda\ (v_1 \ldots v_n)\ \mathit{call}) \rrbracket, \hat{\rho}')}_{\widehat{\mathit{clo}}} \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \\ &\quad \hat{d}_i \in \hat{\mathcal{A}}(\text{æ}_i, \hat{\rho}, \hat{\sigma}) \qquad \hat{t}' = \widehat{\mathit{tick}}(\widehat{\mathit{clo}}, \hat{\varsigma}) \\ &\quad \hat{a}_i = \widehat{\mathit{alloc}}(v_i, \hat{t}') \qquad \hat{\rho}'' = \hat{\rho}'[v_i \mapsto \hat{a}_i] \\ &\quad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \{\hat{d}_i\}]. \end{aligned}$$

2.4.1 Example: k-CFA-style context-sensitivity

The introduction of this component makes it possible to model k-CFA [22] by defining times to be sequences of up to k call sites and addresses to be pairs of variables and call sites:

$$\begin{aligned} \widehat{\mathit{Time}}_{k\text{-CFA}} &= \mathsf{Call}^{\leqslant k} \\ \widehat{\mathit{Addr}}_{k\text{-CFA}} &= \mathsf{Var} \times \widehat{\mathit{Time}}_{k\text{-CFA}}, \end{aligned}$$

and using corresponding definitions

$$\begin{aligned} \widehat{\mathit{tick}}_{k\text{-CFA}}(\widehat{\mathit{clo}}, (\mathit{call}, \ldots, \hat{t})) &= \lfloor \mathit{call} : \hat{t} \rfloor_k \\ \widehat{\mathit{alloc}}_{k\text{-CFA}}(v, (\ldots, \hat{t}')) &= (v, \hat{t}') \end{aligned}$$

where $\lfloor \cdot \rfloor_k$ limits its argument to at most length k.

3. Abstracting through a Monad

In flow analysis, interaction with the store determines essential properties of the analysis. Thus, by abstracting away interaction with the store through a monad, we introduce an abstraction layer for these essential properties.

In pure functional programming, the introduction of store-passing style to mimic side effects is an anti-pattern. The established remedy for this anti-pattern is the use of monadic form in

conjunction with a state-transformer monad. We can apply this remedy to our semantics as well.

To make the presentation unambiguous (and executable), we first transliterate the abstract semantics (for k-CFA at the moment) into Haskell, starting with the syntax for CPS:

```
type Var    = String
data Lambda = [Var] ⇒ CExp deriving (Eq, Ord)
data AExp   = Ref Var
            | Lam Lambda   deriving (Eq, Ord)
data CExp   = Call AExp [AExp]
            | Exit         deriving (Eq, Ord)
```

The set Call from the definition of the state-space corresponds to the type $CExp$ in the implementation. The type for the state-space is a 4-tuple:

```
type Σ      = (CExp, Env, Store, Time)
type k ⇀ v  = Map k v
type Env    = Var ⇀ Addr
type Store  = Addr ⇀ P Val
data Val    = Clo (Lambda, Env)
              deriving (Eq, Ord)
type Addr   = (Var, Time)
type Time   = [CExp]
```

where we utilize Haskell's unicode support to keep the implementation and the math as close as possible (*e.g.*, writing $\mathcal{P}\ Val$ instead of $Set\ Val$). The transition relation becomes a function $next$ on states into lists of states:

```
next :: Σ → [Σ]
next ς@(Call f aes, ρ, σ, t) = [(call, ρ'', σ', t') |
  proc@(Clo (vs ⇒ call, ρ')) ← Set.toList (arg (f, ρ, σ)),
  let t'  = tick (proc, ς)
      as  = [alloc (v, t', proc, ς) | v ← vs]
      ds  = [arg (æ, ρ, σ) | æ ← aes]
      ρ'' = ρ' // [v ⟹ a | v ← vs | a ← as]
      σ'  = σ ⊔ [a ⟹ d | a ← as | d ← ds]]
next ς = [ς]
```

The function arg is the transliteration of the argument evaluator, $\hat{\mathcal{A}}$. The function $tick$ is the transliteration of the $\widehat{tick}$ function. And, the function $alloc$ is the transliteration of the $\widehat{alloc}$ function. Finally, the utility infix operator $(//) :: Ord\ k \Rightarrow (k \rightharpoonup v) \to [(k, v)] \to (k \rightharpoonup v)$ is used to update a map with a list of key-value pairs, and $(\Longrightarrow)$ is just a synonym for pair constructor.

3.1 Capturing nondeterminism in the monad

The function $next$ uses the list comprehension notation to more closely mimic the formalism. The list comprehension notation is syntactic sugar for the list comprehension monad.

We can tiptoe into monadic normal form by expanding the list comprehension $[(call, \rho'', \sigma', t') \mid ...]$ into its monadic form:

```
mnext :: Σ → [Σ]
mnext ς@(Call f aes, ρ, σ, t) = do
  proc@(Clo (vs ⇒ call, ρ')) ← Set.toList (arg (f, ρ, σ))
  let t'  = tick (proc, ς)
      as  = [alloc (v, t', proc, ς) | v ← vs]
      ds  = [arg (æ, ρ, σ) | æ ← aes]
      ρ'' = ρ' // [v ⟹ a | v ← vs | a ← as]
      σ'  = σ ⊔ [a ⟹ d | a ← as | d ← ds]
  return (call, ρ'', σ', t')
mnext ς = return ς
```

Since function evaluation is the only source of nondeterminism, we can clean up these semantics by creating a new function fun as a special version of arg for evaluating functions:

```
fun :: (AExp, Env, Store) → [Val]
fun = Set.toList ∘ arg
```

Thus, we can rewrite $mnext$:

```
mnext :: Σ → [Σ]
mnext ς@(Call f aes, ρ, σ, t) = do
  proc@(Clo (vs ⇒ call, ρ')) ← fun (f, ρ, σ)
  ···    -- the rest is unchanged
```

To reformulate this function fully into monadic normal form, we must refactor and recurry fun, arg, $tick$, $alloc$ to return singleton lists as well (hence fun and arg now have the same type):

```
fun   :: (Env, Store) → AExp → [Val]
arg   :: (Env, Store) → AExp → [Val]
tick  :: Val → State → [Time]
alloc :: (Time, Val, State) → Var → [Addr]
```

which allows us to rewrite $mnext$ in monadic normal form, replacing list comprehensions with a standard function $mapM :: Monad\ m \Rightarrow (a \to m\ b) \to [a] \to m\ [b]$ from a monadic toolset.

```
mnext :: Σ → [Σ]
mnext ς@(Call f aes, ρ, σ, t) = do
  proc@(Clo (vs ⇒ call, ρ')) ← fun (ρ, σ) f
  t' ← tick proc ς
  let as = mapM (alloc (t', proc, ς)) vs
      ds = mapM (arg (ρ, σ)) aes
  ···    -- the rest is unchanged
```

3.2 Pulling the store into the monad

In the previous section, we refactored the abstract semantics so that nondeterminism is explicitly threaded through the list monad. In this section, we subtly reformulate the semantics in terms of a $CPSInterface$ type class that hides interaction with the store, abstracting over five functions that form a *semantic interface* of the CPS calculus:

```
class Monad m ⇒ CPSInterface m where
  fun   :: Env → AExp → m Val
  arg   :: Env → AExp → m Val
  (↦)   :: Addr → Val → m ()
  alloc :: Time → Var → m Addr
  tick  :: Val → PΣ → m Time
```

where the type $P\Sigma$ encodes a partial state (with no store):

$$P\Sigma = (CExp, Env, Time)$$

Under this interface, we can separate the specification of the abstract semantics from the specific details of how it interacts with the store:

```
mnext :: (CPSInterface m) ⇒ PΣ → m PΣ
mnext ps@(Call f aes, ρ, t) = do
  proc@(Clo (vs ⇒ call, ρ')) ← fun ρ f
  t'  ← tick proc ps
  as ← mapM (alloc t') vs
  ds ← mapM (arg ρ) aes
  let ρ'' = ρ' // [v ⟹ a | v ← vs | a ← as]
  sequence [a ↦ d | a ← as | d ← ds]
  return (call, ρ'', t')
mnext ς = return ς
```

3.3 Pulling time into the monad

Like the store, time-stamps were another *ad hoc* addition to the original semantics to engineer them into a form favorable for static analysis. We can lift time-stamps out of partial states and into monads as well, so that $P\Sigma = (CExp, Env)$.

The analysis monad can now assume internal access to the current time, which simplifies the interface:

```
class Monad m ⇒ CPSInterface m where
  fun   :: Env → AExp → m Val
  arg   :: Env → AExp → m Val
  (↦)   :: Addr → Val → m ()
  alloc :: Var → m Addr
  tick  :: Val → PΣ → m ()
```

Because the allocator *alloc* can assume access to time (and the store) inside the monad, it no longer needs to take it as a parameter, leaving the variable to be bound as its *only remaining parameter*. The simplification in the analysis monad is reflected in a simplification of the definition for transition function *mnext* as well, as time is no longer a component of the abstract states datatype $P\Sigma$:

```
mnext :: (CPSInterface m) ⇒ PΣ → m PΣ
mnext ps@(Call f aes, ρ) = do
  proc@(Clo (vs ⇒ call, ρ')) ← fun ρ f
  tick proc ps
  as ← mapM alloc vs
  ···    -- the rest is unchanged
```

3.4 Abstracting over addresses

At this point, we have a monadically abstracted abstracted[3] abstract machine. Plugging in a monad controls nondeterminism, context and access to the store. However, we still have one component left to be abstracted.

Until now, we have assumed that abstract addresses are represented by a fixed datatype. In practice, the nature of the addresses determines the polyvariance and context-sensitivity of the analysis, since they usually divide bindings according to different contexts (also known as *contours* [22]). For example, Shivers' 1CFA allocates distinct contexts for each call site. Lakhotia *et al.* [12] introduce ℓ-contexts to build a static analysis for obfuscated x86 binaries, employing finite sequences of unique enclosed function calls. Finally, one can take a bounded set of naturals $\{n \in \mathbb{N} \mid n \leq N\}$ for some N as contexts, which will give a good precision for sufficiently big N.

Thus, we must abstract over the structure of addresses in order to cover at least all the options from above. Unfortunately, addresses form a part of the state-space as a codomain of *Env*; thus, we need to parameterize our semantic domains by the type of addresses—a:

```
type PΣ a   = (CExp, Env a)
type Env a  = Var ⇀ a
data Val a  = Clo (Lambda, Env a)
              deriving (Eq, Ord)
type Store a = a ⇀ P (Val a)
```

With this shift, we are no longer attached to the k-CFA-like representation of addresses, as both *Addr* and *Time* are gone.

Of course, the signature of the *CPSInterface* monad and *mnext* change accordingly, although the body of *mnext* remains unchanged. The final definition of the *CPSInterface* class and the function *mnext* are represented in Figure 2 and are not going to change in the remainder of our story. What *is* going to change is the implementation of the semantic interface *CPSInterface* as well as the choice of the underlying monad for the analysis logic.

[3] This is not a typo.

```
class Monad m ⇒ CPSInterface m a where
  fun   :: Env a → AExp → m (Val a)
  arg   :: Env a → AExp → m (Val a)
  (↦)   :: a → Val a → m ()
  alloc :: Var → m a
  tick  :: Val a → PΣ a → m ()
mnext :: CPSInterface m a ⇒ PΣ a → m (PΣ a)
mnext ps@(Call f aes, ρ) = do
  proc@(Clo (vs ⇒ call', ρ')) ← fun ρ f
  tick proc ps
  as ← mapM alloc vs
  ds ← mapM (arg ρ) aes
  let ρ'' = ρ' // [v ⟹ a | v ← vs | a ← as]
  sequence [a ↦ d | a ← as | d ← ds]
  return (call', ρ'')
mnext ς = return ς
```

Figure 2: Semantic interface and a small-step semantics of CPS in a monadic form.

4. Recovering a Concrete Interpreter

With non-determinism, interaction with the store and time pulled into an analysis monad, and addresses abstracted, our semantics has become highly parameterized. As a result, we can easily recover a concrete interpreter as a "sanity" check for adequacy. With the standard *IO* monad as an underlying analysis monad, we can use the "real" heap as the store and implement mutable references as a datatype *IOAddr* containing essentially a pointer:

```
data IOAddr = IOAddr {lookup :: IORef (Val IOAddr)}
```

We will also need two simple administrative functions in order to write to and read from *IOAddr*s:

```
readIOAddr :: IOAddr → IO (Val IOAddr)
readIOAddr = readIORef ∘ lookup

writeIOAddr :: IOAddr → Val IOAddr → IO ()
writeIOAddr = writeIORef ∘ lookup
```

The implementation of the *CPSInterface* type class directly mimics the concrete semantics defined in Section 2.2:

```
instance CPSInterface IO IOAddr where
  fun ρ (Lam l)  = return $ Clo (l, ρ)
  fun ρ (Ref v)  = readIOAddr (ρ ! v)

  arg ρ (Lam l)  = return $ Clo (l, ρ)
  arg ρ (Ref v)  = readIOAddr (ρ ! v)

  addr ↦ v       = writeIOAddr addr v
  alloc v        = liftM IOAddr $ newIORef ⊥
  tick _ _       = return ()
```

When a new address is allocated, we pass the function *newIORef* $\bot$ as the initial value of the new reference, since the actual value to be bound at this address is not yet defined at the moment of allocation. The function *tick* is a no-op: in the real world, time advances without our help.

With this interpretation of the semantic interface, the concrete interpreter is simply a recursively defined driver loop that iterates the semantic transition function *mnext* until an *Exit* state is reached.

```
interpret :: CExp → IO (PΣ IOAddr)
interpret e = go (e, Map.empty)
  where go :: (PΣ IOAddr) → IO (PΣ IOAddr)
        go s = do s' ← mnext s
                  case s' of x@(Exit, _) → return x
                             y           → go y
```

5. Recovering a Collecting Semantics

In this section, we recall key notions from lattice theory and abstract interpretation, crucial for constructing an interpreter for concrete or abstract small-step collecting semantics of a program. We proceed by translating the theory into programs in Haskell and demonstrate an implementation of a simple collecting semantics.

5.1 Basics of lattice theory and abstract interpretation

A *complete lattice* $\langle C; \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ is a partial order $\langle C; \sqsubseteq \rangle$ such that there exists a least upper bound (or join) $\sqcup S$ and a greatest lower bound (or meet) $\sqcap S$ of all subsets $S \subseteq C$. In particular $\sqcup C = \top$ and $\sqcap C = \bot$.

A point x is a fixed point of a function f if $f(x) = x$. Given two partial orders, $\langle C, \sqsubseteq \rangle$ and $\langle A, \leq \rangle$, a function f of type $C \to A$ is monotone if $\forall x, y : x \sqsubseteq y \implies f(x) \leq f(y)$. By the Knaster-Tarski fixed-point theorem a monotone functional f over a complete lattice has a *least fixed point* $\mathrm{lfp}_{\sqsubseteq} f = \sqcap\{x \mid f(x) \sqsubseteq x\}$. Algorithmically the least fixed point of a monotone function f over a complete lattice of finite height can be computed by Kleene iteration: $\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq f^3(\bot) \sqsubseteq \ldots$ since

$$\mathrm{lfp}_{\sqsubseteq} f = \bigsqcup_{i \geq 0} f^i(\bot). \tag{1}$$

A *Galois connection* is a pair of functions α, γ connecting two partial orders $\langle C, \sqsubseteq \rangle$ and $\langle A, \leq \rangle$, such that $\forall a, c : \alpha(c) \leq a \iff c \sqsubseteq \gamma(a)$. We typeset Galois connections as: $\langle C, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \leq \rangle$.

Given a Galois connection $\langle C, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \leq \rangle$ and a monotone function $F_c : C \to C$, an *abstract* function F_a can be constructed as $\alpha \circ F_c \circ \gamma \dot{\leq} F_a$. Therefore, by the *fixed-point transfer theorem* [5], we have $\alpha(\mathrm{lfp}\ F_c) \leq \mathrm{lfp}\ F_a$ when F_a is monotone.

The reachable states collecting semantics for a set of program states Σ, a transition function $(\rightsquigarrow) \subseteq \Sigma \times \Sigma$, and a set of initial states $\Sigma_0 \subseteq \Sigma$ is defined on a complete lattice $\langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle$ as a least fixed point of a monotone functional $\mathcal{F}$, where

$$\mathcal{F}(\mathcal{X}) = \Sigma_0 \cup \left\{ s' \mid s \in \mathcal{X} \ \wedge\ s \rightsquigarrow s' \right\}. \tag{2}$$

$\mathcal{F}$ is uncomputable in general, which makes it a usual starting point for construction of an abstract, monotone transition function $\widehat{\mathcal{F}}$, such that $\alpha \circ \mathcal{F} \circ \gamma \dot{\leq} \widehat{\mathcal{F}}$ and computing an approximate fixed point as $\mathrm{lfp}\, \widehat{\mathcal{F}}$.

In the remainder of the section we will systematically abstract over computation of a small-step collecting semantics for CPS.

5.2 Implementing lattices and fixed point computations

We define a type class for a lattice following its algebraic definition:

```
class Lattice a where
  ⊥   :: a
  ⊤   :: a
  (⊑) :: a → a → Bool
  (⊔) :: a → a → a
  (⊓) :: a → a → a
```

We provide the following instances of *Lattice* for the standard container types that we use heavily for systematic abstraction of abstract machines [15]: unit, pairs, powersets and maps:

```
instance Lattice ()
instance (Lattice a, Lattice b) ⇒ Lattice (a, b)
instance (Ord s, Eq s) ⇒ Lattice (P s)
instance (Ord k, Lattice v) ⇒ Lattice (k ⇀ v)
```

The monadic interpreter from Figure 2, which we presented at the end of Section 3, is reminiscent of the definition (2) of the collecting semantics. However, the fact it uses a monad and is not an endo-function, prevents us from using it directly as a relation $(\rightsquigarrow)$ in a functional $\mathcal{F}$. What we can do is to explicitly make a separation of concerns between *mnext* as an implementation of an abstract transition function and a computation of a least fixed point. We define the type class *Collecting* to let the implementor of the analysis define the logic of initiating the semantics (*i.e.*, providing an initial set Σ_0 from (2)) and "making a step":

```
class Collecting m a fp | fp → a, fp → m where
  applyStep :: (a → m a) → fp → fp
  inject    :: a → fp
```

The functional dependencies $fp \to a$ and $fp \to m$ state that the choice of a domain fp determines in which monad m the step function should be interpreted [11], as well as what the transition's domain and co-domain a should be.[4] The computation of the collecting semantics as a least fixed point can be then defined directly from the Kleene iteration (1):

```
kleeneIt :: (Lattice a) ⇒ (a → a) → a
kleeneIt f = loop ⊥
  where loop c = let c' = f c in
                 if c' ⊑ c then c else loop c'
```

Finally, the collecting semantics (2) translates gracefully to the following function, mapping a transition function *step* and an initial state c to a result of the analysis.

```
exploreFP :: (Lattice fp, Collecting m a fp) ⇒
             (a → m a) → a → fp
exploreFP step c = kleeneIt F
  where F s = inject c ⊔ applyStep step s
```

It is now straightforward to implement a function that, given an expression, runs the analysis:

```
runAnalysis :: (CPSInterface m a, Lattice fp,
                Collecting m (PΣ a) fp) ⇒
                CExp → fp
runAnalysis e = exploreFP mnext (e, Map.empty)
```

The signature of *runAnalysis* outlines precisely the three degrees of freedom that can be changed in order to obtain different collecting semantics:

1. A monad, accounting for nondeterminism and passing analysis-specific state components (*i.e.*, time and store),
2. An implementation of the semantic interface of a language (*e.g.*, the one in Figure 2), and
3. The analysis lattice and an implementation of a fixed point computation that extracts the result of a single step from the analysis monad and augments the lattice argument appropriately.

5.3 *StorePassing*—a simple implementation of a collecting semantics for CPS

At this point we are ready to reconstruct a simple abstract interpreter computing a collecting, store-passing semantics of CPS by

[4] Alternatively, we could use associated types to express the same sort of dependencies [3].

gradually instantiating three main aspects of the analysis, outlined at the end of the previous section.

5.3.1 A two-level analysis monad

We start from constructing a monad for a simple collecting, store-passing semantics by employing standard monad transformers $StateT$ and the list monad:

$$\textbf{type}\ StorePassing\ s\ g = StateT\ g\ (StateT\ s\ [])$$

The analysis parameters of type s and g carry state components. They carry the store and the analysis' *guts* respectively, where the latter can contain for example a "time" value.

In a desugared form, the type $StorePassing$ is equivalent to the functional type $g \to s \to [((a, g), s)]$ for some a, g and s, so the stack representation of a monad should be read "inside-out". That is, a value of type $StorePassing$ produces a set (represented by a list) of results of type a, coupled with components of type g and s.

5.3.2 A simple implementation of a CPS semantic interface

To turn the $StorePassing$ monad into an interpretation of the semantics, we have to implement the $CPSInterface$ type class. So far, we choose to implement addresses as Haskell $Integer$s for simplicity. The store is represented as a map from integer addresses to sets of values. The implementation of the semantic interface is provided below.

$$\begin{aligned}
&\textbf{instance}\ CPSInterface\\
&\quad (StorePassing\ (Store\ Integer)\ Integer)\ Integer\\
&\quad \textbf{where}\\
&\qquad fun\ \rho\ (Lam\ l) = return\ \$\ Clo\ (l, \rho)\\
&\qquad fun\ \rho\ (Ref\ v) = lift\ \$\ getsNDSet\ \$\ \lambda\sigma \to \sigma\ !\ (\rho\ !\ v)\\
&\qquad arg\ \rho\ (Lam\ l) = return\ \$\ Clo\ (l, \rho)\\
&\qquad arg\ \rho\ (Ref\ v) = lift\ \$\ getsNDSet\ \$\ \lambda\sigma \to \sigma\ !\ (\rho\ !\ v)\\
&\qquad a \mapsto d \quad\ = lift\ \$\ modify\ \$\\
&\qquad\qquad\qquad Map.insert\ a\ (singleton\ d)\\
&\qquad alloc\ v \quad\ = gets\ id\\
&\qquad tick\ proc\ ps = modify\ \$\ \lambda t \to t + 1
\end{aligned}$$

The function $getsNDSet$ is a crux of handling non-determinism in a monadic implementation of the analysis as a stateful monad. It allows one to examine the state, get multiple results and non-deterministically choose one. Its signature needs explanation:

$$\begin{aligned}
&getsNDSet :: (MonadPlus\ m, MonadState\ s\ m) \Rightarrow\\
&\qquad (s \to Set\ a) \to m\ a
\end{aligned}$$

However, the type class constraints before $\Rightarrow$ provide a clue. One can see that m is required to be a state monad, *i.e.*, carry an implicit state component of type s, which can be accessed and modified. The argument type $s \to Set\ a$ accounts for the non-deterministic result of type a that might come out of examining the state s. Finally, $MonadPlus\ \ m$ is a constraint that ensures that m has a notion of non-deterministic choice, so the obtained results can be combined in a wrapped result $m\ \ a$. What comes as a nice surprise is that $StorePassing$ is an instance of both $MonadPlus$ and $MonadState$.

lift is a standard Haskell function for explicit management of the monadic stack, which is used explicitly to disambiguate the targets of accesses to the monad stack [13, 21]. In our implementation of $CPSInterface$ the outermost state is reserved to carry "guts" (*i.e.*, the time component), so we need to employ *lift* to access the store σ, located on a "second level" of the monad stack. Finally, $gets$ and $modify$ are standard higher-order functions that allow one to examine and modify internal state of the monad.

$$\begin{aligned}
&gets\ \ \ :: MonadState\ s\ m \Rightarrow (s \to a) \to m\ a\\
&modify :: MonadState\ s\ m \Rightarrow (s \to s) \to m\ ()
\end{aligned}$$

For instance, the implementation of $tick$ modifies the time component on the first level of the monadic stack (therefore, no explicit lifting is required).[5]

5.3.3 Computing a collecting semantics of CPS

The last missing ingredient we need to compute the collecting semantics of CPS is the definition of a fixed point computation that uses the $StorePassing$ monad. We can reach a fix-point of type $\mathcal{P}\ ((P\Sigma\ \ a, g), s)$ step by step with the following definitions of $applyStep$ and $inject$.

$$\begin{aligned}
&\textbf{instance}\ (Ord\ s, Ord\ a, Ord\ g, HasInitial\ g, Lattice\ s) \Rightarrow\\
&\qquad Collecting\ (StorePassing\ s\ g)\\
&\qquad\qquad (P\Sigma\ a)\\
&\qquad\qquad (\mathcal{P}\ ((P\Sigma\ a, g), s))\ \textbf{where}\\
&\quad inject\ p = singleton\ \$\ ((p, initial), \bot)\\
&\quad applyStep\ step\ fp = joinWith\ runStep\ fp\ \textbf{where}\\
&\qquad runStep\ ((\varsigma, t), s) =\\
&\qquad\quad Set.fromList\ \$\ runStateT\ (runStateT\ (step\ \varsigma)\ t)\ s
\end{aligned}$$

The implementation of $inject$ instruments a provided state with initial "guts", defined by the value $initial$ of the class $HasInitial$ for g and the lattice minimum $\bot$ for s, and wraps it into a singleton set. The class $HasInitial$ is defined as

$$\textbf{class}\ HasInitial\ g\ \textbf{where}\ initial :: g$$

and its implementation for $Integer$s is trivial (*e.g.*, $initial = 0$).

The most interesting element of the implementation is the utility function $joinWith$, defined as follows:

$$\begin{aligned}
&joinWith :: (Lattice\ a) \Rightarrow\\
&\qquad (b \to a) \to Set\ b \to a\\
&joinWith\ f = Set.foldr\ ((\sqcup) \circ f)\ \bot
\end{aligned}$$

That is, given a set of values of type b and a function f of type $b \to a$ for a lattice a, $joinWith$ traverses the structure, applying f to each of its leaf elements and combines the results using the lattice join ($\sqcup$).

In the definition of $applyStep$, $joinWith$ takes a function that simply passes the state ς and the components t and s to the provided function $step$, runs a monad and collects the result into a set. This function is applied to *all* states in *fp* and the results are joined.

All ingredients to run the analysis are now in place, and all we need to do is to use the function $runAnalysis$ from Section 5.2 to compute the result:

$$\begin{aligned}
&exp :: CExp = ...\\
&runAnalysis\ exp :: \mathcal{P}\ ((P\Sigma\ Integer, Integer), Store\ Integer)
\end{aligned}$$

6. Monadic Parameters for Abstract Abstract Machines

In Section 5 we have demonstrated how to restore a simple collecting semantics from the monadically-parametrized semantic interface, presenting a $StorePassing$ monad and an analysis with domain $\mathcal{P}\ ((P\Sigma\ Integer, Integer), Store\ Integer)$. More complex analyses differ from this simple analysis in a number of aspects. In this section, we will discuss how our $StorePassing$ monad and $CPSInterface$ and $Collecting$ instances can be abstracted further to accomodate this. Specifically, we show how to control polyvariance, store representation, abstract counting, abstract garbage collection and store cloning.

[5] In the present implementation, we allow the time component to grow infinitely for simplicity, so in principle some implementation of the analysis may not terminate, which can be restricted by modifying the function $tick$.

6.1 Controlling polyvariance

So far, we have used Haskell's *Integer*s as abstract addresses in the collecting semantics. To allow experimentation with different addresses, we introduce the following class that provides a uniform view on addresses which may (optionally) incorporate context:

$$\begin{array}{l}\textbf{class}\ (\mathit{Ord}\ a, \mathit{Eq}\ a) \Rightarrow \mathit{Addressable}\ a\ c \mid c \to a\ \textbf{where}\\ \quad \tau_0 \quad :: c\\ \quad \mathit{valloc} \quad :: \mathit{Var} \to c \to a\\ \quad \mathit{advance} :: \mathit{Val}\ a \to P\Sigma\ a \to c \to c\end{array}$$

The type class *Addressable* has two type parameters: a for actual addresses and c for the type of the context. Contexts unambiguously define the nature of addresses. The type class contains three functions. τ_0 generates an initial context, used to instantiate the *HasInitial* class from Section 5.3.3. The function *valloc*, given a variable name and a context, allocates a new address. Finally, the function *advance* has the opportunity to internalize components of the partial state within the monad, based on the function called, the current state processed, and a given context.

A meaningful instance of the *Addressable* type class uses a combination of time-stamps, represented by lists of calls of length bounded by some k, and addresses that simply pair the time-stamps with variable names. This defines an analysis' *polyvariance*:

$$\begin{array}{ll}\textbf{data}\ \mathit{KTime} = \mathit{KCalls}\ [\mathit{CExp}] & \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord})\\ \textbf{data}\ \mathit{KAddr} = \mathit{KBind}\ \mathit{Var}\ \mathit{KTime} & \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord})\end{array}$$

Here is the instance of *Addressable* for this pair, relying on the auxiliary class *KCFA* for generic controlling of polyvariance:

$$\begin{array}{l}\textbf{class}\ \mathit{KCFA}\ a\ \textbf{where}\\ \quad \mathit{getK} :: a \to \mathit{Int}\\ \textbf{instance}\ \mathit{Addressable}\ \mathit{KAddr}\ \mathit{KTime}\ \textbf{where}\\ \quad \tau_0 \quad = \mathit{KCalls}\ [\,]\\ \quad \mathit{valloc}\ v\ t = \mathit{KBind}\ v\ t\\ \quad \mathit{advance}\ \mathit{proc}\ (\mathit{call}, \rho)\ t@(\mathit{KCalls}\ \mathit{calls})\\ \qquad = \mathit{KCalls}\ \$\ (\mathit{getK}\ t)\ (\mathit{call} : \mathit{calls})\end{array}$$

The *CPSInterface* instance for *StorePassing* requires a small change in order to accomodate polyvariance:

$$\begin{array}{l}\textbf{instance}\ (\mathit{Addressable}\ a\ t) \Rightarrow\\ \quad \mathit{CPSInterface}\ (\mathit{StorePassing}\ (\mathit{Store}\ a)\ t)\ a\\ \quad \textbf{where}\\ \quad \mathit{alloc}\ v \quad = \mathit{gets}\ (\mathit{valloc}\ v)\\ \quad \mathit{tick}\ \mathit{proc}\ \mathit{ps} = \mathit{modify}\ \$\ \lambda t \to \mathit{advance}\ \mathit{proc}\ \mathit{ps}\ t\\ \quad \cdots \qquad \text{-- the rest is unchanged}\end{array}$$

Accroding to the *A Posteriori* soundness theorem of Might and Manolios [17], *any* allocation policy for a non-deterministic abstract interpreter (which our analysis is a particular case of) leads to a *sound* abstraction of a concrete store-based collecting semantics that uses uniques addresses for each allocation (*e.g.*, integers, as in the example in Section 5.3.2). Thus, abstracting over addresses yields a family of sound abstract interpreters and requires *no* change in the semantics interface (Figure 2).

6.2 Abstracting over the store component

The store component is essential for efficient implementation of the analysis but can also itself be a source of valuable measurements (*e.g.*, computing the *flows-to* information), so our next step is to make an analysis store-generic. We do so by defining a *StoreLike* class enabling creation of initial store, binding, update, and lookup as well as providing a mechanism to clean the store up:

$$\begin{array}{l}\textbf{class}\ (\mathit{Eq}\ a, \mathit{Lattice}\ s, \mathit{Lattice}\ d) \Rightarrow\\ \qquad \mathit{StoreLike}\ a\ s\ d \mid s \to a, s \to d\ \textbf{where}\\ \quad \sigma_0 \quad :: s\\ \quad \mathit{bind} \quad :: s \to a \to d \to s\\ \quad \mathit{replace} \quad :: s \to a \to d \to s\\ \quad \mathit{fetch} \quad :: s \to a \to d\\ \quad \mathit{filterStore} :: s \to (a \to \mathit{Bool}) \to s\end{array}$$

The *StoreLike* class binds together three components: addresses (a), the store implementation itself (s) and the store co-domain (d). To save space, we refer the reader to our public implementation for implementations of *StoreLike* instances for *Store* a.

Again, we need small changes in the *CPSInterface* instance for *StorePassing* in order to abstract over stores:

$$\begin{array}{l}\textbf{instance}\ (\mathit{Addressable}\ a\ t, \mathit{StoreLike}\ a\ s\ (\mathcal{P}\ (\mathit{Val}\ a))) \Rightarrow\\ \quad \mathit{CPSInterface}\ (\mathit{StorePassing}\ s\ t)\ a\\ \quad \textbf{where}\\ \quad \mathit{fun}\ \rho\ (\mathit{Ref}\ v) = \mathit{lift}\ \$\ \mathit{getsNDSet}\ \$\ \mathit{flip}\ \mathit{fetch}\ (\rho\,!\,v)\\ \quad \mathit{arg}\ \rho\ (\mathit{Ref}\ v) = \mathit{lift}\ \$\ \mathit{getsNDSet}\ \$\ \mathit{flip}\ \mathit{fetch}\ (\rho\,!\,v)\\ \quad a \mapsto d = \mathit{lift}\ \$\ \mathit{modify}\ \$\ \lambda\sigma \to \mathit{bind}\ \sigma\ a\ (\mathit{singleton}\ d)\\ \quad \cdots \qquad \text{-- the rest is unchanged}\end{array}$$

6.3 Controlling abstract counting

Abstract counting is a technique to track how many times an abstract resource has been allocated [18]. It provides a simple but powerful way of bounding cardinalities over abstractions. Specifically, it bounds them in a way that it enables *must*-alias analysis in imperative languages or environment analysis in higher-order languages. Counting enables additional shape analyses or predicate abstractions to be layered on top of an existing analysis [16]. This, in turn, enables an analysis' client to perform environmental analysis to perform advanced optimization, such as super β-inlining [14].

To introduce an abstract counter for CPS, we need to make a small addition to the state-space:

$$\begin{aligned}\hat{\varsigma} \in \hat{\Sigma} &= \mathit{Call} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{Count}} \times \widehat{\mathit{Time}}\\ \hat{\mu} \in \widehat{\mathit{Count}} &= \widehat{\mathit{Addr}} \to \widehat{\mathbb{N}}\\ \hat{n} \in \widehat{\mathbb{N}} &= \{0, 1, \infty\}\,.\end{aligned}$$

The abstract transition relation changes correspondingly to take possible changes of $\hat{\mu}$ into account:

$$\overbrace{(\llbracket(f\ \mathit{ae}_1 \ldots \mathit{ae}_n)\rrbracket, \hat{\rho}, \hat{\sigma}, \hat{\mu}, \hat{t})}^{\hat{\varsigma}} \rightsquigarrow (\mathit{call}, \hat{\rho}'', \hat{\sigma}', \hat{\mu}', \hat{t}'), \text{ where}$$
$$\underbrace{(\llbracket(\lambda\ (v_1 \ldots v_n)\ \mathit{call})\rrbracket, \hat{\rho}')}_{\widehat{clo}} \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})$$

$$\begin{array}{lcllcl}\hat{d}_i & \in & \hat{\mathcal{A}}(\mathit{ae}_i, \hat{\rho}, \hat{\sigma}) & \hat{\rho}'' & = & \hat{\rho}'[v_i \mapsto \hat{a}_i]\\ \hat{t}' & = & \widehat{\mathit{tick}}(\widehat{clo}, \hat{\varsigma}) & \hat{\sigma}' & = & \hat{\sigma} \sqcup [\hat{a}_i \mapsto \{\hat{d}_i\}]\\ \hat{a}_i & = & \widehat{\mathit{alloc}}(v_i, \hat{t}') & \hat{\mu}' & = & \hat{\mu} \mathbin{\dot{\oplus}} [\hat{a}_i \mapsto 1]\end{array}$$

The operator $\dot{\oplus}$ is the point-wise lifted natural abstraction of addition over $\widehat{\mathbb{N}}$. In fact, the structure of the set $\widehat{\mathbb{N}}$ can vary depending on the requirements for the analysis' results. The only requirement is that $\widehat{\mathbb{N}}$ should be a lattice. For instance, in a degenerate case, one can turn abstract counting off by setting $\widehat{\mathbb{N}} = \{\infty\}$.

Since abstract counts are modified in tandem with the abstract store, counting and dependent enhancements like strong update can be hidden entirely within and tuned from the monad. We need no refactoring of the state-space or the semantics needed to introduce abstract counting, thanks to the introduction of the class *StoreLike* in Section 6.2. All we need to do is supply a different instance of store, namely, a "counting" one. First, we define a datatype for $\widehat{\mathbb{N}}$, make it a lattice and define abstract addition:

$$\begin{array}{l}\textbf{data}\ AbsNat = AZero \mid AOne \mid AMany \\ \quad \textbf{deriving}\ (Ord, Eq, Show) \\ (\oplus) :: AbsNat \to AbsNat \to AbsNat \\ AZero \oplus n \quad = n \\ n \quad \oplus AZero = n \\ n \quad \oplus m \quad = AMany \\ \textbf{instance}\ Lattice\ AbsNat\ \textbf{where}\ \cdots\end{array}$$

Second, we define a type class for abstract counter, a counting store and make the latter an instance of the former:

$$\begin{array}{l}\textbf{class}\ StoreLike\ a\ s\ d \Rightarrow ACounter\ a\ s\ d\ \textbf{where} \\ \quad count :: s \to a \to AbsNat \\ \textbf{type}\ CountingStore\ a\ d = a \rightharpoonup (d, AbsNat) \\ \textbf{instance}\ (Ord\ a, Lattice\ d) \Rightarrow \\ \qquad ACounter\ a\ (CountingStore\ a\ d)\ d\ \textbf{where} \\ \quad count\ \sigma\ a = snd\ \$\ \sigma\ !\ a\end{array}$$

Because the counter is parameterized over addresses, it too is independent of specific semantics, and in fact can be used with any other semantics.

For a full implementation of $StoreLike\ a\ (CountingStore\ a\ d)\ d$, we invite the reader to puzzle through it or reference our public implementation. Once the instance is provided, a $CountingStore$ can be directly plugged into the $StorePassing$, so the abstract counting does not require any changes in the analysis logic, implicitly adding the component $\widehat{Count}$ to $\hat{\Sigma}$.

6.4 Controlling abstract garbage collection

Abstract garbage collection [18] is a store-sensitive analysis technique that prunes unreachable structure (exactly as ordinary garbage collection does). The net effect is an often dramatic increase in precision as well as a corresponding drop in analysis time. The technique is defined in terms of touchability of a value by a binding, the adjacency of bindings and reachability of bindings from some entity. In essence, abstract *garbage collection* stands for finding the set of reachable bindings for a particular state and restricting the domain of the store σ to solely these bindings.

The abstract bindings $\hat{\mathcal{T}}(\text{æ}, \hat{\rho})$, touched by some abstract closure pair $(\text{æ}, \hat{\rho})$ are defined as follows:

$$\begin{array}{rcl}\hat{\mathcal{T}}(\text{æ}, \hat{\rho}) & = & \{\hat{\rho}(v) : v \in free(\text{æ})\} \\ \hat{\mathcal{T}}\ \{(\text{æ}_1, \hat{\rho}_1), \ldots, (\text{æ}_n, \hat{\rho}_n)\} & = & \hat{\mathcal{T}}(\text{æ}_1, \hat{\rho}_1) \cup \ldots \cup \hat{\mathcal{T}}(\text{æ}_n, \hat{\rho}_n)\end{array}$$

We extend the notion of touching to call sites and abstract states:

$$\begin{array}{rcl}\hat{\mathcal{T}}((f\ \text{æ}_1 \ldots \text{æ}_n), \hat{\rho}) & = & \hat{\mathcal{T}}(f, \hat{\rho}) \cup \hat{\mathcal{T}}(\text{æ}_1, \hat{\rho}) \cup \ldots \hat{\mathcal{T}}(\text{æ}_n, \hat{\rho}) \\ \hat{\mathcal{T}}(call, \hat{\rho}, \hat{\sigma}, \hat{t}) & = & \hat{\mathcal{T}}(call, \hat{\rho})\end{array}$$

and define the abstract adjacency relation for bindings:

$$\hat{a} \hat{\rightsquigarrow}_{\hat{\sigma}} \hat{a}' \iff \hat{a}' \in \hat{\mathcal{T}}(\hat{\sigma}(\hat{a}))$$

Next, the abstract reachable-bindings function

$$\hat{\mathcal{R}} : \widehat{\Sigma} \to \mathcal{P}\left(\widehat{Addr}\right)$$

for a given abstract state $\hat{\varsigma}$ computes the set of *reachable* bindings as all bindings we can reach from $\hat{\varsigma}$ with chains of $\hat{\rightsquigarrow}_{\hat{\sigma}}$ links:

$$\hat{\mathcal{R}}(\hat{\varsigma}) = \left\{\hat{a}' : \hat{a} \in \hat{\mathcal{T}}(\hat{\varsigma}) \text{ and } \hat{a}\ \hat{\rightsquigarrow}^*_{\hat{\sigma}_{\hat{\varsigma}}}\ \hat{a}'\right\}.$$

We define the abstract Garbage Collection function, $\hat{\Gamma} : \widehat{\Sigma} \to \widehat{\Sigma}$ that removes unreachable bindings from the domain of $\hat{\sigma}$:

$$\hat{\Gamma}(call, \hat{\rho}, \hat{\sigma}, \hat{t}) = \quad (call, \hat{\rho}, \hat{\sigma}|\hat{\mathcal{R}}(\hat{\varsigma}), \hat{t}),$$

where the vertical bar '|' operator denotes map restriction, *i.e.*, $f|X$ is the function f defined at most over elements in the set X.

Finally, using the function $\hat{\Gamma}$, we can define the alternate, GC abstract transition rule, $\rightsquigarrow_{\hat{\Gamma}}$, so the abstract transition becomes:

$$\frac{\hat{\Gamma}(\hat{\varsigma}) \rightsquigarrow \hat{\varsigma}'}{\hat{\varsigma} \rightsquigarrow_{\hat{\Gamma}} \hat{\varsigma}'} \qquad \text{(STEP-GC)}$$

From the implementor's point of view, an abstract garbage collector modifies the *internal* part of the state, *i.e.*, an abstract store, by removing unreachable addresses. Therefore, it is natural to define abstract garbage collection abstractly as an operation in the analysis monad:

$$\begin{array}{l}\textbf{class}\ Monad\ m \Rightarrow GarbageCollector\ m\ a\ \textbf{where} \\ \quad gc :: a \to m\ () \\ \quad gc\ \_ = return\ () \quad \text{-- default implementation}\end{array}$$

The function gc takes a partial state as a parameter and returns a monad operation. We also supply a *default* implementation of the function gc as a no-op.

The structure of the class $StoreLike$ makes it easy to implement a garbage collector thanks to the function *filterStore* (Section 6.2). Weaving the GC into the semantics requires only little change in the fixed point computation for $StorePassing$:

$$\begin{array}{l}applyStep\ step = \\ \quad joinWith\ (\lambda((\varsigma, t), s) \to \\ \quad \ldots runStateT\ (\textbf{do}\ \{\varsigma' \leftarrow step\ \varsigma; gc\ \varsigma'; return\ \varsigma'\})\ t \ldots)\end{array}$$

6.5 Controlling store-cloning

By default, the abstracted abstract machine approach to static analysis yields a heap-(/store-)cloning analysis: every state contains a store. Ordinarily, store-cloning should be reserved for situations in which the extra precision benefit the target applications.

However, for an analysis implemented this way it can take time exponential in the size of the input program when computing the reachable states of the abstracted machine [19]. The standard technique to reduce the complexity is to employ widening in the form of Shivers' single-threaded store [22]. To use a single-threaded store, we have to reconsider the abstract evaluation function itself. Instead of seeing it as a function that returns the set of reachable states, it is a function that returns a pair, consisting of a set of partial states and a single globally approximating store.

Although this change requires a significant reworking in the definition of the semantics, it is quite easy to implement in our framework, since the store is *not* a component of the program states, but rather an element supplied by an analysis monad. One can also notice that the new semantics can be captured by establishing the following Galois connection:

$$\langle \mathcal{P}(\widehat{\Sigma_t} \times \widehat{Store}), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{P}(\widehat{\Sigma_t}) \times \widehat{Store}, \subseteq \rangle, \qquad (3)$$

where $\widehat{\Sigma_t} = \mathsf{Call} \times \widehat{Env} \times \widehat{Time}$. Bounding the space of addresses (Section 6.1) implies finiteness of both lattices involved in the Galois connection (3), which means that both α and γ are computable. It is straightforward to express them in Haskell:

$$\begin{array}{l}alpha :: \quad (Lattice\ s, Ord\ a, Ord\ g) \Rightarrow \\ \qquad \mathcal{P}\ ((P\Sigma\ a, g), s) \to (\mathcal{P}\ (P\Sigma\ a, g), s) \\ alpha = joinWith\ (\lambda((p, g), \sigma) \to (singleton\ (p, g), \sigma)) \\ gamma :: (Ord\ a, Ord\ g, Ord\ s) \Rightarrow \\ \qquad (\mathcal{P}\ (P\Sigma\ a, g), s) \to \mathcal{P}\ ((P\Sigma\ a, g), s) \\ gamma\ (states, \sigma) = Set.map\ (\lambda(p, g) \to ((p, g), \sigma))\ states\end{array}$$

In words, $alpha$ combines together all per-state store components (given the stores form a lattice, hence $Lattice\ s$). Conversely, $gamma$ spreads the store σ among all provided states.

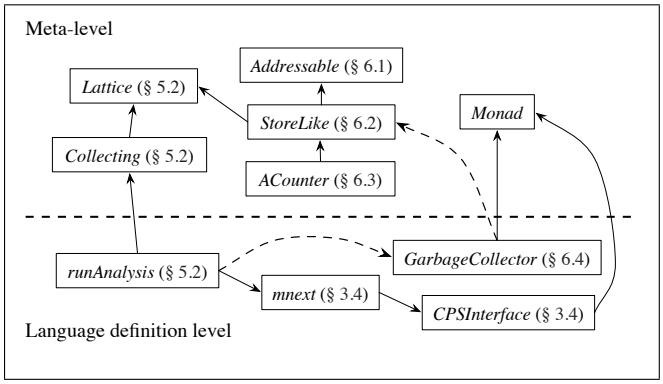

Figure 3: Overview of the framework concepts

The easiest way to construct a single-store analysis is to redefine the instance of $Collecting$, taking $F_c = applyStep'\ step$:[6]

$$\begin{aligned}&\textbf{instance}\ (Ord\ g, Ord\ a, Ord\ s, Lattice\ s, HasInitial\ g) \Rightarrow\\ &\quad Collecting\ (StorePassing\ s\ g)\\ &\qquad\qquad (P\Sigma\ a)\\ &\qquad\qquad (\mathcal{P}\ (P\Sigma\ a, g), s)\ \textbf{where}\\ &\quad inject\ a\ \ \ \ \ \ \ \ = (singleton\ (a, initial), \bot)\\ &\quad applyStep\ step = alpha \circ (applyStep'\ step) \circ gamma\end{aligned}$$

7. Pulling It All Together

A high-level overview of the described framework is given in Figure 3. The *concepts* in the upper half are defined on the *meta-level*, *i.e.*, *can be* implemented in a way so they might be reused by different languages and analyses. The lower half describes language-dependent concepts, which rely, in particular, on the fixed syntax, semantics and the structure of values. Solid arrows denote type dependencies between *instances* of the components, and dashed arrows are for the optional logical dependencies, which are, however, not strictly enforced by types. Curiously, the diagram confirms our point that store logic and abstract counting strategy are orthogonal to the analysis implementation (Sections 6.2 and 6.3).

As opposite to the abstract *concepts*, it is much harder to provide a straightforward hierarchy of actual concept *implementations* with strict "level" distinctions. For instance, one can encode the store and monad logic directly into the semantic interface definition, moving it, therefore to the bottom part of the diagram. Alternatively, one can implement the store and the monad independently from the encoding of the semantic interface, just as we did in Section 3, getting as close as possible to the conceptual decomposition. This approach pays off when composing the resulting analysis, as we demonstrate in the following section.

8. Further Examples: k-CFA Family

In this section, we expand our original examples from Section 3 to a family of k-CFA-based abstract interpreters, following the recipe described in Sections 5 and 6. We omit a few tedious implementation details for the sake of brevity; the full development and test programs can be found in the accompanying code repository:

`http://github.com/ilyasergey/monadic-cfa`

[6] We use $applyStep'$ to refer to the definition of $applyStep$ for a domain with per-state store. It is slightly different from the actual implementation in Haskell, as one is required to wrap the "per-state store"-version of $StorePassing$ into a separate datatype in order to make it possible for a type checker to distinguish between two versions of $applyStep$. We elaborate more on this in Section 8.

8.1 A simple abstract interpreter for k-CFA

First, we fix the k-degree of the analysis by instantiating the class $KCFA$ from Section 6.1 for $k = 1$:

$$\begin{aligned}&\textbf{instance}\ KCFA\ KTime\ \textbf{where}\\ &\quad getK = const\ 1\end{aligned}$$

Second, we define the analysis function $analyseKCFA$ by employing the $runAnalysis$ function from Section 5.2:

$$\begin{aligned}&analyseKCFA :: CExp \rightarrow\\ &\quad \mathcal{P}\ ((P\Sigma\ KAddr, KTime), Store\ KAddr)\\ &analyseKCFA = runAnalysis\end{aligned}$$

The refined type of $analyseKCFA$ is of particular interest. After flattening the tuples, one can see that it reflects the abstract domain of the analysis: abstract states coupled with timestamps and abstract stores.

8.2 An abstract interpreter with a shared store

As a next step, we apply the widening strategy via a shared store, as described in Section 6.5. We directly use the definitions of $alpha$ and $gamma$ and redefine the instance of $Collecting$ by providing a new implementation of the $applyStep$ function (Section 5.2). In order to overcome Haskell's conventions for type resolution in the case of the function $applyStep$, we might need to define a *wrapper* record type for the analysis result.

$$\begin{aligned}&\textbf{newtype}\ Wrap\ a\ g\ s =\\ &\quad Wrap\ \{unWrap :: \mathcal{P}\ ((P\Sigma\ a, g), s)\}\ \textbf{deriving}\ Lattice\end{aligned}$$

The definition of $applyStep$ for store-sharing widening looks as follows:

$$\begin{aligned}&applyStep\ step =\\ &\quad alpha \circ unWrap \circ applyStep\ step \circ Wrap \circ gamma\end{aligned}$$

Note, the implementation is *not* recursive, but the inner call to $applyStep$ operates with a different domain. The definition of the analysis function has the same implementation, thanks to the type class-based polymorphism [25], however, its return type is conceptually different, as it accounts for a set of states, coupled with a *single* store:

$$\begin{aligned}&analyseShared :: CExp \rightarrow\\ &\quad (\mathcal{P}\ (P\Sigma\ KAddr, KTime), Store\ KAddr)\\ &analyseShared = runAnalysis\end{aligned}$$

8.3 An abstract interpreter with a counting store

As the analysis is parametrized with the store explicitly, its instrumentation with a counting machinery is trivial: we just replace the second component of the result of the single-store-passing analysis with a specialized counting store (Section 6.3).

$$\begin{aligned}&\textbf{type}\ KCFACountingStore =\\ &\quad CountingStore\ KAddr\ (\mathcal{P}\ (Val\ KAddr))\\ &analyseWithCount :: CExp \rightarrow\\ &\quad (\mathcal{P}\ (P\Sigma\ KAddr, KTime), KCFACountingStore)\\ &analyseWithCount = runAnalysis\end{aligned}$$

9. Related Work and Conclusion

We have illustrated a systematic method for transforming a *concrete* semantics into a monadically-parameterized machine, such that the monad determines the classical properties of an *abstract* analysis. Our work is situated firmly in the abstract interpretation tradition established by Cousot and Cousot [4, 5].

Following a series of complex power-domain constructions, Hudak and Young [10] devised simpler set-based collecting interpretations for both first-order and higher-order functional languages. They furthermore outline how to modify their approach to express *relational* properties. This requires generalizing their collecting interpretation to sets of value-environment pairs, remniscent of our sets-of-states starting point. Like Hudak and Young, our framework can thereby describe relational properties.

The way we factor the concrete semantics is very much in the spirit of Nielson's two-level meta-language [20]. Nielson proposes an abstract interpretation framework based on the idea of decomposing a *denotational* language definition into two parts: a *core semantics*, containing semantic rules and their types, but leaving some function symbols and domain names uninterpreted, and an *interpretation* that fills out the missing definition of function symbol and domain names. thereby allowing alternate non-standard interpretations in addition to the "standard" semantics defining the meaning of programs.

Such a decomposition is formulated in terms of a *two-level* metalanguage, where some types are considered to be 'dynamically'-interpreted,[7] and 'dynamic' functions symbols are represented by combinators, closed over variables of dynamically-interpreted types. This makes it possible to define the "meaning" of dynamic types and combinators in different ways to express a lazy standard semantics, detection of signs, strictness, and liveness on top of the same semantics interface. Also, Nielson further illustrates that both forward and backward analyses, independent-attribute and relational methods can be formulated in terms of the same *core* semantics given different *interpretations*. In contrast, our work focuses on abstractions of the *operational* small-step collecting semantics, since it delivers data for most of the interesting analyses in the setting of a higher-order language. In a sense, the functions from the class $CPSInterface$, such as fun, arg and others, play an analogous role to Nielson's dynamic function symbols, and type parameters such as a, s or g are the dynamic types.

Our framework gains from giving function symbols monadic types: unlike Nielson's framework, where all information about properties of interest is captured by the treatment of values of specific types, our implementation also allows us to track *temporal* properties of execution and tweak interpretation properties depending on the context of execution. Moreover, the sharing of stores as a widening strategy, is trivial to implement in our decomposition. The *well-formedness* of an interpretation in our framework is ensured by the type system of the host language (Haskell, in our case).

The core semantics we present using monads is of a more denotational flavour, as it is expressed by the $CPSInterface$ type class. Recent work by Filinski, however, demonstrates that a complementary, operational representation, is possible using *reflection* and *reification* [8]. This correspondence could be used to translate the monad-based definitions of the semantics functions into specialized operational rules, which were usually hand-crafted [23].

Our publicly available implementation indicates the robustness of the approach, allowing re-use of multiple semantic aspects between different analyses and semantic formalisms.

Acknowledgements We wish to thank Olivier Danvy for discussions on Nielson's work as well as for his hospitality during Sergey and Might's visit to Aarhus University in December 2011, where the idea of the work was discussed for the first time. We are grateful to the PLDI 2013 reviewers for their excellent feedback. The work of Ilya Sergey has been partially supported by EU Marie Curie COFUND Action 291803 "Amarout-II Europe". This research is partially funded by the Research Foundation - Flanders (FWO), and by the Research Fund KU Leuven. Dominique Devriese holds a Ph.D. fellowship of the Research Foundation - Flanders (FWO). Different parts of Might's effort on this work were partially supported by the DARPA programs APAC and CRASH.

[7] *I.e.*, those, which can be given a specific interpretation as a domain or a lattice.

References

[1] M. S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theor. Comput. Sci.*, 342(1):149–172, 2005.

[2] M. Biernacka and O. Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theor. Comput. Sci.*, 375(1-3):76–108, 2007.

[3] M. M. T. Chakravarty, G. Keller, S. L. P. Jones, and S. Marlow. Associated types with class. In *POPL*, 2005.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[5] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.

[6] O. Danvy. A new one-pass transformation into monadic normal form. In *CC*, volume 2622 of *LNCS*, 2003.

[7] O. Danvy. Defunctionalized interpreters for programming languages. In *ICFP*, 2008.

[8] A. Filinski. Monads in action. In *POPL*, 2010.

[9] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*, 1993.

[10] P. Hudak and J. Young. Collecting interpretations of expressions. *ACM Trans. Prog. Lang. Syst.*, 13(2):269–290, 1991.

[11] M. P. Jones. Type Classes with Functional Dependencies. In *ESOP*, volume 1782 of *LNCS*, 2000.

[12] A. Lakhotia, D. R. Boccardo, A. Singh, and A. Manacero, Jr. Context-sensitive analysis of obfuscated x86 executables. In *PEPM*, 2010.

[13] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL*, 1995.

[14] M. Might. *Environment analysis of higher-order languages*. PhD thesis, Georgia Institute of Technology, 2007.

[15] M. Might. Abstract interpreters for free. In *SAS*, volume 6337 of *LNCS*, 2010.

[16] M. Might. Shape analysis in the absence of pointers and structure. In *VMCAI*, volume 5944 of *LNCS*, 2010.

[17] M. Might and P. Manolios. A posteriori soundness for non-deterministic abstract interpretations. In *VMCAI*, volume 5403 of *LNCS*, 2009.

[18] M. Might and O. Shivers. Improving flow analyses via ΓCFA: abstract garbage collection and counting. In *ICFP*, 2006.

[19] M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *PLDI*, 2010.

[20] F. Nielson. Two-level semantics and abstract interpretation. *Theor. Comput. Sci.*, 69(2):117–242, 1989.

[21] T. Schrijvers and B. C. Oliveira. Monads, zippers and views: virtualizing the monad stack. In *ICFP*, 2011.

[22] O. G. Shivers. *Control-flow analysis of higher-order languages or taming lambda*. PhD thesis, Carnegie Mellon University, 1991.

[23] D. Van Horn and M. Might. Abstracting abstract machines. In *ICFP*, 2010.

[24] D. Van Horn and M. Might. Abstracting abstract machines: a systematic approach to higher-order program analysis. *Commun. ACM*, 54(9):101–109, 2011.

[25] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, 1989.