Monadic Abstract Interpreters

Ilya Sergey   Dominique Devriese   Matthew Might

Jan Midtgaard   David Darais   Dave Clarke   Frank Piessens

PLDI 2013

"My life goal: Replace myself with a $\LaTeX$ macro."

Matthew Might

Abstract Interpreters for Free, SAS 2010

M. Might, *“Abstract Interpreters for Free”*

small-step concrete semantics (interpreter)

=>

small-step abstract semantics (analysis)

This Work

This Work

Replace myself with a library of reusable functions.

This Work

small-step concrete semantics implementation

and

small-step abstract semantics implementation

This Work

small-step concrete semantics implementation

and

small-step abstract semantics implementation

(for the price of one + $\mathcal{E}$)

How do you

design

an abstract interpreter?

How do you

implement

an abstract interpreter?

Our perspective

Separate
the interpreter machinery
from a program analysis logic

Separate
the interpreter machinery
from a program analysis logic

and also

Make different aspects of a program analysis
reusable between languages and semantics

Monads for
the separation of concerns

Starting point: Concrete vs. Abstract

Concrete CPS semantics

$$
\begin{aligned}
([\![(f\ æ_1 \ldots æ_n)]\!], \rho, \sigma) &\Rightarrow (call, \rho'', \sigma'), \text{ where} \\
([\![(\lambda\ (v_1 \ldots v_n)\ call)]\!], \rho') &= \mathcal{A}(f, \rho, \sigma) \\
\rho'' &= \rho'[v_i \mapsto a_i] \\
\sigma' &= \sigma[a_i \mapsto \mathcal{A}(æ_i, \rho, \sigma)] \\
a_i &= alloc(v_i, \sigma)
\end{aligned}
$$

where

$$
\begin{aligned}
\mathcal{A}(v, \rho, \sigma) &= \sigma(\rho(v)) \\
\mathcal{A}(lam, \rho, \sigma) &= (lam, \rho)
\end{aligned}
$$

Abstract CPS semantics

$$
\begin{aligned}
([\![(f\ æ_1 \ldots æ_n)]\!], \hat{\rho}, \hat{\sigma}) &\rightsquigarrow (call, \hat{\rho}'', \hat{\sigma}'), \text{ where} \\
([\![(\lambda\ (v_1 \ldots v_n)\ call)]\!], \hat{\rho}') &\in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \\
\hat{\rho}'' &= \hat{\rho}'[v_i \mapsto \hat{a}_i] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(æ_i, \hat{\rho}, \hat{\sigma})] \\
\hat{a}_i &= \widehat{alloc}(v_i, \hat{\sigma})
\end{aligned}
$$

where

$$
\begin{aligned}
\hat{\mathcal{A}}(v, \hat{\rho}, \hat{\sigma}) &= \hat{\sigma}(\hat{\rho}(v)) \\
\hat{\mathcal{A}}(lam, \hat{\rho}, \hat{\sigma}) &= \{(lam, \hat{\rho})\}
\end{aligned}
$$

Similar, but not the same!

$$
\begin{aligned}
(\llbracket(f\ æ_1 \ldots æ_n)\rrbracket, \rho, \sigma) &\Rightarrow (call, \rho'', \sigma'), \text{ where} \\
(\llbracket(\lambda\ (v_1 \ldots v_n)\ call)\rrbracket, \rho') &= \mathcal{A}(f, \rho, \sigma) \\
\rho'' &= \rho'[v_i \mapsto a_i] \\
\sigma' &= \sigma[a_i \mapsto \mathcal{A}(æ_i, \rho, \sigma)] \\
a_i &= alloc(v_i, \sigma)
\end{aligned}
$$

$$
\begin{aligned}
(\llbracket(f\ æ_1 \ldots æ_n)\rrbracket, \hat{\rho}, \hat{\sigma}) &\leadsto (call, \hat{\rho}'', \hat{\sigma}'), \text{ where} \\
(\llbracket(\lambda\ (v_1 \ldots v_n)\ call)\rrbracket, \hat{\rho}') &\in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \\
\hat{\rho}'' &= \hat{\rho}'[v_i \mapsto \hat{a}_i] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(æ_i, \hat{\rho}, \hat{\sigma})] \\
\hat{a}_i &= \widehat{alloc}(v_i, \hat{\sigma})
\end{aligned}
$$

How can we unify
their implementations?

Commonalities

Differences

Commonalities

Commonalities

Shape of the computation

Concrete $\overbrace{([\![(f\ æ_1 \ldots æ_n)]\!], \rho, \sigma)}^{\varsigma} \Rightarrow (call, \rho'', \sigma')$

Abstract $\overbrace{([\![(f\ æ_1 \ldots æ_n)]\!], \hat{\rho}, \hat{\sigma})}^{\hat{\varsigma}} \leadsto (call, \hat{\rho}'', \hat{\sigma}')$

Differences

Differences

Treatment of semantic values

Concrete

$$
\begin{aligned}
([\![(f\ æ_1 \ldots æ_n)]\!], \rho, \sigma) &\Rightarrow (call, \rho'', \sigma'), \text{ where} \\
([\![(\lambda\ (v_1 \ldots v_n)\ call)]\!], \rho') &= \mathcal{A}(f, \rho, \sigma) \\
\rho'' &= \rho'[v_i \mapsto a_i] \\
\sigma' &= \sigma[a_i \mapsto \mathcal{A}(æ_i, \rho, \sigma)] \\
a_i &= alloc(v_i, \sigma)
\end{aligned}
$$

Abstract

$$
\begin{aligned}
([\![(f\ æ_1 \ldots æ_n)]\!], \hat{\rho}, \hat{\sigma}) &\leadsto (call, \hat{\rho}'', \hat{\sigma}'), \text{ where} \\
([\![(\lambda\ (v_1 \ldots v_n)\ call)]\!], \hat{\rho}') &\in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \\
\hat{\rho}'' &= \hat{\rho}'[v_i \mapsto \hat{a}_i] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(æ_i, \hat{\rho}, \hat{\sigma})] \\
\hat{a}_i &= \widehat{alloc}(v_i, \hat{\sigma})
\end{aligned}
$$

Differences

Treatment of semantic values

Concrete

$$([\![(f\ æ_1 \ldots æ_n)]\!], \rho, \sigma) \Rightarrow (call, \rho'', \sigma'), \text{ where}$$
$$([\![(\lambda\ (v_1 \ldots v_n)\ call)]\!], \rho') = \mathcal{A}(f, \rho, \sigma)$$
$$\rho'' = \rho'[v_i \mapsto a_i]$$
$$\sigma' = \sigma[a_i \mapsto \mathcal{A}(æ_i, \rho, \sigma)]$$
$$a_i = alloc(v_i, \sigma)$$

Abstract

$$([\![(f\ æ_1 \ldots æ_n)]\!], \hat{\rho}, \hat{\sigma}) \leadsto (call, \hat{\rho}'', \hat{\sigma}'), \text{ where}$$
$$([\![(\lambda\ (v_1 \ldots v_n)\ call)]\!], \hat{\rho}') \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})$$
$$\hat{\rho}'' = \hat{\rho}'[v_i \mapsto \hat{a}_i]$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(æ_i, \hat{\rho}, \hat{\sigma})]$$
$$\hat{a}_i = \widehat{alloc}(v_i, \hat{\sigma})$$

Abstract Interpreter

Abstract Interpreter

- Forks

Abstract Interpreter

- Forks
- Advances timestamps

Abstract Interpreter

- Forks
- Advances timestamps
- Performs Abstract GC

Abstract Interpreter

- Forks
- Advances timestamps
- Performs Abstract GC
- Keeps track of contexts

Abstract Interpreter

- Forks
- Advances timestamps
- Performs Abstract GC
- Keeps track of contexts
- Makes counting

Abstract Interpreter

- Forks
- Advances timestamps
- Performs Abstract GC
- Keeps track of contexts
- Makes counting

} Computational Effects

Abstract Interpreter

nondeterminism
- Forks

tracing
- Advances timestamps

state modification
- Performs Abstract GC

tracing
- Keeps track of contexts

state modification
- Makes counting

} Computational Effects

Abstract Interpretation as a computational effect

Eugenio Moggi

Notions of computation and monads, *Inf. Comput.*, 1991

... we identify the type *A* with the object of *values* (of type *A*) and obtain the object of *computations* (of type *A*) by applying an unary type-constructor *T* to *A*.
We call *T* a *notion of computation*, since it abstracts away from the type of values computations may produce.

Eugenio Moggi

Notions of computation and monads, *Inf. Comput.*, 1991

... we identify the type *A* with the object of *values* (of type *A*) and obtain the object of *computations* (of type *A*) by applying an unary type-constructor *T* to *A*.
We call *T* a *notion of computation*, since it abstracts away from the type of values computations may produce.

Philip Wadler

Comprehending Monads, LFP, 1991

It is relatively straightforward to adopt Moggi's technique of structuring denotational specifications into a technique for structuring functional programs. This paper presents a simplified version of Moggi's ideas, framed in a way better suited to functional programmers than semanticists; in particular, no knowledge of category theory is assumed.

Philip Wadler

Comprehending Monads, LFP, 1991

It is relatively straightforward to adopt Moggi's technique of structuring denotational specifications into a technique for structuring functional programs. This paper presents a simplified version of Moggi's ideas, framed in a way better suited to functional programmers than semanticists; in particular, no knowledge of category theory is assumed.

Let’s program some semantics in Haskell

Implementing and Refactoring time-stamped *k*-CFA for CPS

$$v \in \mathsf{Var} \text{ is a set of identifiers}$$

$$lam \in \mathsf{Lam} ::= (\lambda\ (v_1 \ldots v_n)\ call)$$

$$f, æ \in \mathsf{AExp} = \mathsf{Var} + \mathsf{Lam}$$

$$call \in \mathsf{Call} ::= (f\ æ_1 \ldots æ_n) + \mathsf{Exit}$$

$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Call} \times \widehat{Env} \times \widehat{Store} \times \widehat{Time}$$

$$\hat{\rho} \in \widehat{Env} = \mathsf{Var} \rightharpoonup \widehat{Addr}$$

$$\hat{\sigma} \in \widehat{Store} = \widehat{Addr} \to \mathcal{P}(\hat{D})$$

$$\hat{d} \in \hat{D} = \widehat{Clo}$$

$$\widehat{clo} \in \widehat{Clo} = \mathsf{Lam} \times \widehat{Env}$$

$$\hat{a} \in \widehat{Addr} = \mathsf{Var} \times \widehat{Time}$$

$$\hat{t} \in \widehat{Time} = \mathsf{Call}^{\leq k}$$

$$
\begin{array}{llll}
\textbf{type}\ \mathit{Var} & = \mathit{String} & & \\
\textbf{data}\ \mathit{Lambda} & = [\,\mathit{Var}\,] \Rightarrow \mathit{CExp} & \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}) & \\
\textbf{data}\ \mathit{AExp} & = \mathit{Ref}\ \ \mathit{Var} & & \\
 & \mid\ \ \mathit{Lam}\ \mathit{Lambda} & \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}) & \\
\textbf{data}\ \mathit{CExp} & = \mathit{Call}\ \mathit{AExp}\ [\,\mathit{AExp}\,] & & \\
 & \mid\ \ \mathit{Exit} & \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}) &
\end{array}
$$

$$
\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Call} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{Time}} \\
\hat{\rho} \in \widehat{\mathit{Env}} &= \mathsf{Var} \rightharpoonup \widehat{\mathit{Addr}} \\
\hat{\sigma} \in \widehat{\mathit{Store}} &= \widehat{\mathit{Addr}} \to \mathcal{P}(\hat{D}) \\
\hat{d} \in \hat{D} &= \widehat{\mathit{Clo}} \\
\widehat{\mathit{clo}} \in \widehat{\mathit{Clo}} &= \mathsf{Lam} \times \widehat{\mathit{Env}} \\
\hat{a} \in \widehat{\mathit{Addr}} &= \mathsf{Var} \times \widehat{\mathit{Time}} \\
\hat{t} \in \widehat{\mathit{Time}} &= \mathsf{Call}^{\leq k}
\end{aligned}
$$

$$
\begin{array}{llll}
\textbf{type}\ \mathit{Var} & = \mathit{String} & & \\
\textbf{data}\ \mathit{Lambda} & = [\mathit{Var}] \Rightarrow \mathit{CExp} & \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}) & \\
\textbf{data}\ \mathit{AExp} & = \mathit{Ref}\ \ \mathit{Var} & & \\
 & \mid\ \ \mathit{Lam}\ \mathit{Lambda} & \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}) & \\
\textbf{data}\ \mathit{CExp} & = \mathit{Call}\ \mathit{AExp}\ [\mathit{AExp}] & & \\
 & \mid\ \ \mathit{Exit} & \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}) &
\end{array}
$$

$$
\begin{array}{ll}
\textbf{type}\ \Sigma & = (\mathit{CExp}, \mathit{Env}, \mathit{Store}, \mathit{Time}) \\
\textbf{type}\ k \rightharpoonup v & = \mathit{Map}\ k\ v \\
\textbf{type}\ \mathit{Env} & = \mathit{Var} \rightharpoonup \mathit{Addr} \\
\textbf{type}\ \mathit{Store} & = \mathit{Addr} \rightharpoonup \mathcal{P}\ \mathit{Val} \\
\textbf{data}\ \mathit{Val} & = \mathit{Clo}\ (\mathit{Lambda}, \mathit{Env}) \\
 & \qquad\quad \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}) \\
\textbf{type}\ \mathit{Addr} & = (\mathit{Var}, \mathit{Time}) \\
\textbf{type}\ \mathit{Time} & = [\mathit{CExp}]
\end{array}
$$

$$(\leadsto) \in \Sigma \to \mathcal{P}(\Sigma)$$

$$\overbrace{(\llbracket (f\ æ_1 \ldots æ_n) \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{t})}^{\hat{\varsigma}} \leadsto (call, \hat{\rho}'', \hat{\sigma}', \hat{t}'), \text{ if}$$

$$\underbrace{(\llbracket (\lambda\ (v_1 \ldots v_n)\ call) \rrbracket, \hat{\rho}')}_{\widehat{clo}} \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})$$

$$\hat{t}' = \widehat{tick}(\widehat{clo}, \hat{\varsigma})$$

$$\hat{a}_i = \widehat{alloc}(v_i, \hat{t}')$$

$$\hat{d}_i \in \hat{\mathcal{A}}(æ_i, \hat{\rho}, \hat{\sigma})$$

$$\hat{\rho}'' = \hat{\rho}'[v_i \mapsto \hat{a}_i]$$

$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \{\hat{d}_i\}]$$

$$next :: \Sigma \to [\Sigma]$$

$$\overbrace{(\llbracket(f\ æ_1 \ldots æ_n)\rrbracket, \hat{\rho}, \hat{\sigma}, \hat{t})}^{\hat{\varsigma}} \rightsquigarrow (call, \hat{\rho}'', \hat{\sigma}', \hat{t}'), \text{ if}$$

$$\underbrace{(\llbracket(\lambda\ (v_1 \ldots v_n)\ call)\rrbracket, \hat{\rho}')}_{\widehat{clo}} \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})$$

$$\hat{t}' = \widehat{tick}(\widehat{clo}, \hat{\varsigma})$$

$$\hat{a}_i = \widehat{alloc}(v_i, \hat{t}')$$

$$\hat{d}_i \in \hat{\mathcal{A}}(æ_i, \hat{\rho}, \hat{\sigma})$$

$$\hat{\rho}'' = \hat{\rho}'[v_i \mapsto \hat{a}_i]$$

$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \{\hat{d}_i\}]$$

$$next :: \Sigma \to [\Sigma]$$

$$\begin{aligned}
next\ \varsigma@(Call\ f\ \ aes, \rho, \sigma, t) = [\,(call, \rho'', \sigma', t')\ |\\
&proc@(Clo\ (vs \Rightarrow call, \rho')) \leftarrow Set.toList\ (arg\ (f, \rho, \sigma)),\\
&\mathbf{let}\ t' \ = tick\ (proc, \varsigma)\\
&\qquad as \ = [\,alloc\ (v, t', proc, \varsigma)\ |\ v \leftarrow vs\,]\\
&\qquad ds \ = [\,arg\ (æ, \rho, \sigma)\ |\ æ \leftarrow aes\,]\\
&\qquad \rho'' = \rho' \,/\!/\, [\,v \Longrightarrow a\ |\ v \leftarrow vs\ |\ a \leftarrow as\,]\\
&\qquad \sigma' \ = \sigma \sqcup [\,a \Longrightarrow d\ |\ a \leftarrow as\ |\ d \leftarrow ds\,]]
\end{aligned}$$

$$next\ \varsigma = [\varsigma]$$

Refactoring Plan

- Capture non-determinism in the monad
- Pull the store into the monad
- Pull the time into the monad
- Abstract over *k*-CFA addresses

Refactoring Plan

➔ • Capture non-determinism in the monad

- Pull the store into the monad
- Pull the time into the monad
- Abstract over *k*-CFA addresses

$$next :: \Sigma \to [\Sigma]$$

$$\begin{aligned}
&next\ \varsigma@(Call\ f\ \ aes, \rho, \sigma, t) = [(call, \rho'', \sigma', t') \mid \\
&\qquad proc@(Clo\ (vs \Rightarrow call, \rho')) \leftarrow Set.toList\ (arg\ (f, \rho, \sigma)), \\
&\qquad \mathbf{let}\ t' \ = tick\ (proc, \varsigma) \\
&\qquad\qquad as \ = [\,alloc\ (v, t', proc, \varsigma) \mid v \leftarrow vs] \\
&\qquad\qquad ds \ = [\,arg\ (\text{æ}, \rho, \sigma) \mid \text{æ} \leftarrow aes] \\
&\qquad\qquad \rho'' = \rho' \mathbin{/\!/} [\,v \Longrightarrow a \mid v \leftarrow vs \mid a \leftarrow as] \\
&\qquad\qquad \sigma' \ = \sigma \sqcup [\,a \Longrightarrow d \mid a \leftarrow as \mid d \leftarrow ds]]
\end{aligned}$$

$$next\ \varsigma = [\varsigma]$$

$$mnext :: \Sigma \to [\Sigma]$$

$$\begin{aligned}
&next\ \varsigma@(Call\ f\ \ aes, \rho, \sigma, t) = [(call, \rho'', \sigma', t')\ |\\
&\quad proc@(Clo\ (vs \Rightarrow call, \rho')) \leftarrow Set.toList\ (arg\ (f, \rho, \sigma)),\\
&\quad \mathbf{let}\ t'\ = tick\ (proc, \varsigma)\\
&\qquad as\ = [\,alloc\ (v, t', proc, \varsigma)\ |\ v \leftarrow vs\,]\\
&\qquad ds\ = [\,arg\ (æ, \rho, \sigma)\ |\ æ \leftarrow aes\,]\\
&\qquad \rho'' = \rho' /\!\!/ [\,v \Longrightarrow a\ |\ v \leftarrow vs\ |\ a \leftarrow as\,]\\
&\qquad \sigma'\ = \sigma \sqcup [\,a \Longrightarrow d\ |\ a \leftarrow as\ |\ d \leftarrow ds\,]]
\end{aligned}$$

$$next\ \varsigma = [\varsigma]$$

$$mnext :: \Sigma \rightarrow [\Sigma]$$

$$\begin{aligned}
&mnext\ \varsigma@(Call\ f\ \ aes, \rho, \sigma, t) = \mathbf{do}\\
&\quad proc@(Clo\ (vs \Rightarrow call, \rho')) \leftarrow Set.toList\ (arg\ (f, \rho, \sigma)),\\
&\quad \mathbf{let}\ t' \ = tick\ (proc, \varsigma)\\
&\qquad as\ = [\,alloc\ (v, t', proc, \varsigma) \mid v \leftarrow vs\,]\\
&\qquad ds\ = [\,arg\ (\text{æ}, \rho, \sigma) \mid \text{æ} \leftarrow aes\,]\\
&\qquad \rho'' = \rho' \mathbin{/\!/} [\,v \Longrightarrow a \mid v \leftarrow vs \mid a \leftarrow as\,]\\
&\qquad \sigma'\ = \sigma \sqcup [\,a \Longrightarrow d \mid a \leftarrow as \mid d \leftarrow ds\,]\\
&\quad return\ (call, \rho'', \sigma', t')
\end{aligned}$$

$$next\ \varsigma = [\varsigma]$$

$$mnext :: \Sigma \to [\Sigma]$$

$$
\begin{aligned}
&mnext\ \varsigma@(Call\ f\ \ aes, \rho, \sigma, t) = \mathbf{do} \\
&\quad proc@(Clo\ (vs \Rightarrow call, \rho')) \leftarrow Set.toList\ (arg\ (f, \rho, \sigma)), \\
&\quad \mathbf{let}\ t' \ = tick\ (proc, \varsigma) \\
&\qquad as = [\,alloc\ (v, t', proc, \varsigma) \mid v \leftarrow vs\,] \\
&\qquad ds = [\,arg\ (\text{æ}, \rho, \sigma) \mid \text{æ} \leftarrow aes\,] \\
&\qquad \rho'' = \rho' \mathbin{/\!\!/} [\,v \Longrightarrow a \mid v \leftarrow vs \mid a \leftarrow as\,] \\
&\qquad \sigma' \ = \sigma \sqcup [\,a \Longrightarrow d \mid a \leftarrow as \mid d \leftarrow ds\,] \\
&\quad return\ (call, \rho'', \sigma', t')
\end{aligned}
$$

$$mnext\ \varsigma = return\ \varsigma$$

Semantic functions

$$
\begin{aligned}
fun &:: (Env, Store) \to AExp \to [\,Val\,] \\
arg &:: (Env, Store) \to AExp \to [\,Val\,] \\
tick &:: Val \to State \to [\,Time\,] \\
alloc &:: (\,Time, Val, State) \to Var \to [\,Addr\,]
\end{aligned}
$$

Semantic functions

$$\left.\begin{array}{l} \mathit{fun} \;\; :: (\mathit{Env}, \mathit{Store}) \to \mathit{AExp} \to [\,\mathit{Val}\,] \\ \mathit{arg} \;\; :: (\mathit{Env}, \mathit{Store}) \to \mathit{AExp} \to [\,\mathit{Val}\,] \end{array}\right\} \quad \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})$$

$$\mathit{tick} \;\; :: \mathit{Val} \to \mathit{State} \to [\,\mathit{Time}\,]$$

$$\mathit{alloc} :: (\mathit{Time}, \mathit{Val}, \mathit{State}) \to \mathit{Var} \to [\,\mathit{Addr}\,]$$

$$mnext :: \Sigma \to [\Sigma]$$

$$
\begin{aligned}
&mnext\ \varsigma@(Call\ f\ \ aes, \rho, \sigma, t) = \mathbf{do}\\
&\quad proc@(Clo\ (vs \Rightarrow call, \rho')) \leftarrow Set.toList\ (arg\ (f, \rho, \sigma))\\
&\quad \mathbf{let}\ t'\ = tick\ (proc, \varsigma)\\
&\qquad\quad as = [\,alloc\ (v, t', proc, \varsigma) \mid v \leftarrow vs\,]\\
&\qquad\quad ds = [\,arg\ (\text{æ}, \rho, \sigma) \mid \text{æ} \leftarrow aes\,]\\
&\qquad\quad \rho'' = \rho' \mathbin{/\!/} [\,v \Longrightarrow a \mid v \leftarrow vs \mid a \leftarrow as\,]\\
&\qquad\quad \sigma' = \sigma \sqcup [\,a \Longrightarrow d \mid a \leftarrow as \mid d \leftarrow ds\,]\\
&\quad return\ (call, \rho'', \sigma', t')
\end{aligned}
$$

$$mnext\ \varsigma = return\ \varsigma$$

$$mnext :: \Sigma \to [\Sigma]$$

$$
\begin{aligned}
&mnext\ \varsigma@(Call\ f\ \ aes, \rho, \sigma, t) = \mathbf{do}\\
&\quad proc@(Clo\ (vs \Rightarrow call, \rho')) \leftarrow\ fun\ (\rho, \sigma)\ f\\
&\quad t' \leftarrow tick\ proc\ \varsigma\\
&\quad \mathbf{let}\ as = mapM\ \ (alloc\ (t', proc, \varsigma))\ vs\\
&\quad\qquad ds = mapM\ \ (arg\ (\rho, \sigma))\ aes\\
&\quad\qquad \rho'' = \rho' /\!/ [v \Longrightarrow a \mid v \leftarrow vs \mid a \leftarrow as]\\
&\quad\qquad \sigma'\ = \sigma \sqcup [a \Longrightarrow d \mid a \leftarrow as \mid d \leftarrow ds]\\
&\quad return\ (call, \rho'', \sigma', t')
\end{aligned}
$$

$$mnext\ \varsigma = return\ \varsigma$$

Refactoring Plan

➔ • Capture non-determinism in the monad
- Pull the store into the monad
- Pull the time into the monad
- Abstract over *k*-CFA addresses

Refactoring Plan

- ✓ Capture non-determinism in the monad
- ➔ Pull the store into the monad
- Pull the time into the monad
- Abstract over *k*-CFA addresses

Semantic functions

$$
\begin{aligned}
fun &:: (Env, Store) \to AExp \to [\,Val\,] \\
arg &:: (Env, Store) \to AExp \to [\,Val\,] \\
tick &:: Val \to State \to [\,Time\,] \\
alloc &:: (Time, Val, State) \to Var \to [\,Addr\,]
\end{aligned}
$$

Semantic functions

$$
\begin{aligned}
fun &:: (Env, Store) \to AExp \to [Val] \\
arg &:: (Env, Store) \to AExp \to [Val] \\
tick &:: Val \to State \to [Time] \\
alloc &:: (Time, Val, State) \to Var \to [Addr]
\end{aligned}
$$

Semantic functions

$$
\begin{aligned}
&\textbf{class } Monad\ m \Rightarrow CPSInterface\ m \textbf{ where}\\
&\quad fun \quad :: Env \to AExp \to m\ Val\\
&\quad arg \quad :: Env \to AExp \to m\ Val\\
&\quad (\mapsto) \ :: Addr \to Val \to m\ ()\\
&\quad alloc :: Time \to Var \to m\ Addr\\
&\quad tick \quad :: Val \to P\Sigma \to m\ Time
\end{aligned}
$$

Semantic functions

$$
\begin{aligned}
&\textbf{class } \mathit{Monad}\ m \Rightarrow \mathit{CPSInterface}\ m \textbf{ where}\\
&\quad \mathit{fun}\ \ :: \mathit{Env} \to \mathit{AExp} \to m\ \mathit{Val}\\
&\quad \mathit{arg}\ \ :: \mathit{Env} \to \mathit{AExp} \to m\ \mathit{Val}\\
&\quad (\mapsto)\ :: \mathit{Addr} \to \mathit{Val} \to m\ ()\\
&\quad \mathit{alloc} :: \mathit{Time} \to \mathit{Var} \to m\ \mathit{Addr}\\
&\quad \mathit{tick}\ \ :: \mathit{Val} \to P\Sigma \to m\ \mathit{Time}
\end{aligned}
$$

Semantic functions

$$
\begin{array}{l}
\textbf{class}\ Monad\ m \Rightarrow CPSInterface\ m\ \textbf{where} \\
\quad fun\ \ :: Env \to AExp \to m\ Val \\
\quad arg\ \ :: Env \to AExp \to m\ Val \\
\quad (\mapsto)\ :: Addr \to Val \to m\ () \\
\quad alloc :: Time \to Var \to m\ Addr \\
\quad tick\ \ :: Val \to P\Sigma \to m\ Time
\end{array}
$$

$\underbrace{(CExp, Env, Time)}$ - "partial state"

Semantic Interface

$$\begin{array}{l}
\textbf{class}\ Monad\ m \Rightarrow CPSInterface\ m\ \textbf{where} \\
\quad fun \ \ :: Env \to AExp \to m\ Val \\
\quad arg \ \ :: Env \to AExp \to m\ Val \\
\quad (\mapsto) :: Addr \to Val \to m\ () \\
\quad alloc :: Time \to Var \to m\ Addr \\
\quad tick \ \ :: Val \to P\Sigma \to m\ Time
\end{array}$$

$\overbrace{(CExp, Env, Time)}$ - "partial state"

$$mnext :: \Sigma \rightarrow [\Sigma]$$

$$
\begin{aligned}
&mnext\ \varsigma@(Call\ f\ \ aes, \rho, \sigma, t) = \mathbf{do} \\
&\quad proc@(Clo\ (vs \Rightarrow call, \rho')) \leftarrow fun\ (\rho, \sigma)\ f \\
&\quad t' \leftarrow tick\ proc\ \varsigma \\
&\quad \mathbf{let}\ as = mapM\ (alloc\ (t', proc, \varsigma))\ vs \\
&\quad\quad\quad ds = mapM\ (arg\ (\rho, \sigma))\ aes \\
&\quad\quad\quad \rho'' = \rho' \mathbin{/\!/} [v \Longrightarrow a \mid v \leftarrow vs \mid a \leftarrow as] \\
&\quad\quad\quad \sigma' = \sigma \sqcup [a \Longrightarrow d \mid a \leftarrow as \mid d \leftarrow ds] \\
&\quad return\ (call, \rho'', \sigma', t')
\end{aligned}
$$

$$mnext\ \varsigma = return\ \varsigma$$

$$mnext :: (CPSInterface\ m) \Rightarrow P\Sigma \to m\ P\Sigma$$

$$
\begin{aligned}
& mnext\ \varsigma@(Call\ f\ aes, \rho, \sigma, t) = \mathbf{do} \\
& \quad proc@(Clo\ (vs \Rightarrow call, \rho')) \leftarrow fun\ (\rho, \sigma)\ f \\
& \quad t' \leftarrow tick\ proc\ \varsigma \\
& \quad \mathbf{let}\ as = mapM\ (alloc\ (t', proc, \varsigma))\ vs \\
& \quad \phantom{\mathbf{let}}\ ds = mapM\ (arg\ (\rho, \sigma))\ aes \\
& \quad \phantom{\mathbf{let}}\ \rho'' = \rho' \mathbin{/\!/} [v \Longrightarrow a \mid v \leftarrow vs \mid a \leftarrow as] \\
& \quad \phantom{\mathbf{let}}\ \sigma' = \sigma \sqcup [a \Longrightarrow d \mid a \leftarrow as \mid d \leftarrow ds] \\
& \quad return\ (call, \rho'', \sigma', t')
\end{aligned}
$$

$$mnext\ \varsigma = return\ \varsigma$$

$$mnext :: (CPSInterface\ m) \Rightarrow P\Sigma \to m\ P\Sigma$$

$$\begin{aligned}
&mnext\ \varsigma@(Call\ f\ aes, \rho, \sigma, t) = \mathbf{do}\\
&\quad proc@(Clo\ (vs \Rightarrow call, \rho')) \leftarrow fun\ (\rho, \sigma)\ f\\
&\quad t' \leftarrow tick\ proc\ \varsigma\\
&\quad as \leftarrow mapM\ (alloc\ t')\ vs\\
&\quad ds \leftarrow mapM\ (arg\ \rho)\ aes\\
&\quad \mathbf{let}\ \rho'' = \rho' \mathbin{/\!/} [v \Longrightarrow a \mid v \leftarrow vs \mid a \leftarrow as]\\
&\quad sequence\ [a \mapsto d \mid a \leftarrow as \mid d \leftarrow ds]\\
&\quad return\ (call, \rho'', \sigma', t')\\
&mnext\ \varsigma = return\ \varsigma
\end{aligned}$$

Refactoring Plan

- ✓ Capture non-determinism in the monad
- ➔ Pull the store into the monad
- Pull the time into the monad
- Abstract over *k*-CFA addresses

Refactoring Plan

✓ • Capture non-determinism in the monad

✓ • Pull the store into the monad

➔ • Pull the time into the monad

• Abstract over *k*-CFA addresses

Semantic Interface

$$
\begin{aligned}
&\mathbf{class}\ Monad\ m \Rightarrow CPSInterface\ m\ \mathbf{where}\\
&\quad fun\ \ :: Env \to AExp \to m\ Val\\
&\quad arg\ \ :: Env \to AExp \to m\ Val\\
&\quad (\mapsto)\ :: Addr \to Val \to m\ ()\\
&\quad alloc :: Time \to Var \to m\ Addr\\
&\quad tick\ \ :: Val \to P\Sigma \to m\ Time
\end{aligned}
$$

Semantic Interface

$$
\begin{array}{l}
\textbf{class } Monad\ m \Rightarrow CPSInterface\ m\ \textbf{where} \\
\quad \begin{array}{ll}
fun & :: Env \rightarrow AExp \rightarrow m\ Val \\
arg & :: Env \rightarrow AExp \rightarrow m\ Val \\
(\mapsto) & :: Addr \rightarrow Val \rightarrow m\ () \\
alloc & :: Var \rightarrow m\ Addr \\
tick & :: Val \rightarrow P\Sigma \rightarrow m\ ()
\end{array}
\end{array}
$$

Semantic Interface

$$
\begin{array}{l}
\textbf{class } \mathit{Monad}\ m \Rightarrow \mathit{CPSInterface}\ m\ \textbf{where} \\
\quad \mathit{fun} \ :: \mathit{Env} \to \mathit{AExp} \to m\ \mathit{Val} \\
\quad \mathit{arg} \ :: \mathit{Env} \to \mathit{AExp} \to m\ \mathit{Val} \\
\quad (\mapsto) \ :: \mathit{Addr} \to \mathit{Val} \to m\ () \\
\quad \mathit{alloc} :: \mathit{Var} \to m\ \mathit{Addr} \\
\quad \mathit{tick} \ :: \mathit{Val} \to P\Sigma \to m\ ()
\end{array}
$$

$\overbrace{(\mathit{CExp}, \mathit{Env})}$ - "*pure* partial state"

$$mnext :: (CPSInterface\ m) \Rightarrow P\Sigma \rightarrow m\ P\Sigma$$

$$mnext\ \varsigma@(Call\ f\ aes, \rho, \sigma, t) = \mathbf{do}$$
$$proc@(Clo\ (vs \Rightarrow call, \rho')) \leftarrow fun\ (\rho, \sigma)\ f$$
$$t' \leftarrow tick\ proc\ ps$$
$$as \leftarrow mapM\ (alloc\ t')\ vs$$
$$ds \leftarrow mapM\ (arg\ \rho)\ aes$$
$$\mathbf{let}\ \rho'' = \rho' \mathbin{/\!/} [v \Longrightarrow a \mid v \leftarrow vs \mid a \leftarrow as]$$
$$sequence\ [a \mapsto d \mid a \leftarrow as \mid d \leftarrow ds]$$
$$return\ (call, \rho'', \sigma', t')$$

$$mnext\ \varsigma = return\ \varsigma$$

$$mnext :: (CPSInterface\ m) \Rightarrow P\Sigma \rightarrow m\ P\Sigma$$

$$\begin{aligned}
&mnext\ \varsigma@(Call\ f\ \ aes, \rho, \sigma, t) = \mathbf{do} \\
&\quad proc@(Clo\ (vs \Rightarrow call, \rho')) \leftarrow\ fun\ (\rho, \sigma)\ f \\
&\quad tick\ proc\ ps \\
&\quad as \leftarrow mapM\ alloc\ vs \\
&\quad ds \leftarrow mapM\ (arg\ \rho)\ aes \\
&\quad \mathbf{let}\ \rho'' = \rho' \mathbin{/\!/} [\,v \Longrightarrow a \mid v \leftarrow vs \mid a \leftarrow as] \\
&\quad sequence\ [\,a \mapsto d \mid a \leftarrow as \mid d \leftarrow ds] \\
&\quad return\ (call, \rho'', \sigma', t') \\
&mnext\ \varsigma = return\ \varsigma
\end{aligned}$$

Refactoring Plan

- ✓ Capture non-determinism in the monad
- ✓ Pull the store into the monad
- ➔ Pull the time into the monad
- Abstract over *k*-CFA addresses

Refactoring Plan

- ✓ • Capture non-determinism in the monad
- ✓ • Pull the store into the monad
- ✓ • Pull the time into the monad
- → • Abstract over *k*-CFA addresses

$$
\begin{aligned}
\textbf{type } P\Sigma \quad &= \quad (CExp, Env) \\
\textbf{type } Env \quad &= \quad Var \rightharpoonup Addr \\
\textbf{data } Val \quad &= \quad Clo\ (Lambda, Env) \\
\textbf{type } Store \quad &= \quad Addr \rightharpoonup \mathcal{P}(Val)
\end{aligned}
$$

$$
\begin{aligned}
\textbf{type } Addr \quad &= (Var, Time) \\
\textbf{type } Time \quad &= [CExp]
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{type}\ P\Sigma &&= (\mathit{CExp}, \mathit{Env}) \\
&\textbf{type}\ \mathit{Env} &&= \mathit{Var} \rightharpoonup \mathit{Addr} \\
&\textbf{data}\ \mathit{Val} &&= \mathit{Clo}\ (\mathit{Lambda}, \mathit{Env}) \\
&\textbf{type}\ \mathit{Store} &&= \mathit{Addr} \rightharpoonup \mathcal{P}(\mathit{Val})
\end{aligned}
$$

$$
\begin{aligned}
\textbf{type}\ P\Sigma\ a &= (CExp, Env\, a)\\
\textbf{type}\ Env\ a &= Var \rightharpoonup a\\
\textbf{data}\ Val\ a &= Clo\ (Lambda, Env\ a)\\
\textbf{type}\ Store\ a &= a \rightharpoonup \mathcal{P}(Val\ a)
\end{aligned}
$$

Refactoring Plan

✓ • Capture non-determinism in the monad

✓ • Pull the store into the monad

✓ • Pull the time into the monad

→ • Abstract over *k*-CFA addresses

Refactoring Plan

- ✓ Capture non-determinism in the monad
- ✓ Pull the store into the monad
- ✓ Pull the time into the monad
- ✓ Abstract over *k*-CFA addresses

Refactoring Plan

- `List` Capture non-determinism in the monad
- `State` Pull the store into the monad
- `Writer` Pull the time into the monad
- Abstract over *k*-CFA addresses

Monadic Small-Step Transition

$$
\begin{array}{l}
mnext :: CPSInterface\ m\ a \Rightarrow P\Sigma\ a \rightarrow m\ (P\Sigma\ a) \\
mnext\ ps@(Call\ f\ aes, \rho) = \mathbf{do} \\
\quad proc@(Clo\ (vs \Rightarrow call', \rho')) \leftarrow fun\ \rho\ f \\
\quad tick\ proc\ ps \\
\quad as \leftarrow mapM\ alloc\ vs \\
\quad ds \leftarrow mapM\ (arg\ \rho)\ aes \\
\quad \mathbf{let}\ \rho'' = \rho' \mathbin{/\!/} [v \Longrightarrow a \mid v \leftarrow vs \mid a \leftarrow as] \\
\quad sequence\ [a \mapsto d \mid a \leftarrow as \mid d \leftarrow ds] \\
\quad return\ (call', \rho'') \\
mnext\ \varsigma = return\ \varsigma
\end{array}
$$

Monadic Small-Step Transition

$$
\begin{array}{l}
mnext :: CPSInterface\ m\ a \Rightarrow P\Sigma\ a \rightarrow m\ (P\Sigma\ a) \\
mnext\ ps@(Call\ f\ aes, \rho) = \mathbf{do} \\
\quad proc@(Clo\ (vs \Rightarrow call', \rho')) \leftarrow fun\ \rho\ f \\
\quad tick\ proc\ ps \\
\quad as \leftarrow mapM\ alloc\ vs \\
\quad ds \leftarrow mapM\ (arg\ \rho)\ aes \\
\quad \mathbf{let}\ \rho'' = \rho' \mathbin{/\!/} [v \Longrightarrow a \mid v \leftarrow vs \mid a \leftarrow as] \\
\quad sequence\ [a \mapsto d \mid a \leftarrow as \mid d \leftarrow ds] \\
\quad return\ (call', \rho'') \\
mnext\ \varsigma = return\ \varsigma
\end{array}
$$

Fixed

Semantic Interface

$$
\begin{aligned}
&\textbf{class}\ Monad\ m \Rightarrow CPSInterface\ m\ a\ \textbf{where}\\
&\quad fun\ \ :: Env\ a \rightarrow AExp \rightarrow m\ (Val\ a)\\
&\quad arg\ \ :: Env\ a \rightarrow AExp \rightarrow m\ (Val\ a)\\
&\quad (\mapsto)\ :: a \rightarrow Val\ a \rightarrow m\ ()\\
&\quad alloc :: Var \rightarrow m\ a\\
&\quad tick\ \ :: Val\ a \rightarrow P\Sigma\ a \rightarrow m\ ()
\end{aligned}
$$

The Semantic Interface

$$
\begin{aligned}
&\mathbf{class}\ Monad\ m \Rightarrow CPSInterface\ m\ a\ \mathbf{where}\\
&\quad fun \ \ :: Env\ a \to AExp \to m\ (Val\ a)\\
&\quad arg \ \ :: Env\ a \to AExp \to m\ (Val\ a)\\
&\quad (\mapsto)\ :: a \to Val\ a \to m\ ()\\
&\quad alloc :: Var \to m\ a\\
&\quad tick \ \ :: Val\ a \to P\Sigma\ a \to m\ ()
\end{aligned}
$$

The Semantic Interface

$$
\begin{aligned}
&\textbf{class } \mathit{Monad}\ m \Rightarrow \mathit{CPSInterface}\ m\ a\ \textbf{where}\\
&\quad \mathit{fun}\ \ :: \mathit{Env}\ a \to \mathit{AExp} \to m\ (\mathit{Val}\ a)\\
&\quad \mathit{arg}\ \ :: \mathit{Env}\ a \to \mathit{AExp} \to m\ (\mathit{Val}\ a)\\
&\quad (\mapsto)\ :: a \to \mathit{Val}\ a \to m\ ()\\
&\quad \mathit{alloc} :: \mathit{Var} \to m\ a\\
&\quad \mathit{tick}\ \ :: \mathit{Val}\ a \to P\Sigma\ a \to m\ ()
\end{aligned}
$$

Needs to be instantiated

So what now?

Instantiating Monadic Semantics

Instance I: Shallow Concrete Interpreter

Instance I: Shallow Concrete Interpreter

`IO`

+

Semantic Interface Implementation

+

Standard driver loop machinery

Addresses

$$\mathbf{data}\ IOAddr = IOAddr\ \{lookup :: IORef\ (Val\ IOAddr)\}$$

Read / Write

$$readIOAddr :: IOAddr \to IO\ (Val\ IOAddr)$$
$$readIOAddr = readIORef \circ lookup$$

$$writeIOAddr :: IOAddr \to Val\ IOAddr \to IO\ ()$$
$$writeIOAddr = writeIORef \circ lookup$$

Semantic Functions for Concrete Semantics

$$
\begin{array}{l}
\textbf{instance}\ \mathit{CPSInterface\ IO\ IOAddr}\ \textbf{where} \\
\quad \mathit{fun}\ \rho\ (\mathit{Lam}\ l) = \mathit{return}\ \$\ \mathit{Clo}\ (l, \rho) \\
\quad \mathit{fun}\ \rho\ (\mathit{Ref}\ \ v)\ = \mathit{readIOAddr}\ (\rho\ !\ v) \\
\quad \mathit{arg}\ \rho\ (\mathit{Lam}\ l) = \mathit{return}\ \$\ \mathit{Clo}\ (l, \rho) \\
\quad \mathit{arg}\ \rho\ (\mathit{Ref}\ \ v)\ = \mathit{readIOAddr}\ (\rho\ !\ v) \\
\quad \mathit{addr} \mapsto v \qquad = \mathit{writeIOAddr}\ \mathit{addr}\ v \\
\quad \mathit{alloc}\ v \qquad\quad = \mathit{liftM}\ \ \mathit{IOAddr}\ \$\ \mathit{newIORef}\ \bot \\
\quad \mathit{tick}\ \_\ \_ \qquad\quad = \mathit{return}\ ()
\end{array}
$$

Semantic Functions for Concrete Semantics

$$\overbrace{}^{Monad}$$

$$
\begin{aligned}
&\textbf{instance } CPSInterface\ \overbrace{IO}\ IOAddr\ \textbf{where} \\
&\quad fun\ \rho\ (Lam\ l) = return\ \$\ Clo\ (l, \rho) \\
&\quad fun\ \rho\ (Ref\ \ v) = readIOAddr\ (\rho\ !\ v) \\
&\quad arg\ \rho\ (Lam\ l) = return\ \$\ Clo\ (l, \rho) \\
&\quad arg\ \rho\ (Ref\ \ v) = readIOAddr\ (\rho\ !\ v) \\
&\quad addr \mapsto v \qquad\quad = writeIOAddr\ \ addr\ \ v \\
&\quad alloc\ v \qquad\qquad = liftM\ \ IOAddr\ \$\ newIORef\ \bot \\
&\quad tick\ \_\ \_ \qquad\quad\ = return\ ()
\end{aligned}
$$

Semantic Functions for Concrete Semantics

$$
\begin{aligned}
&\textbf{instance } \mathit{CPSInterface}\ \overbrace{\mathit{IO}}^{\mathit{Monad}}\ \overbrace{\mathit{IOAddr}}^{\mathit{Addr}}\ \textbf{where} \\
&\quad \mathit{fun}\ \rho\ (\mathit{Lam}\ l) = \mathit{return}\ \$\ \mathit{Clo}\ (l, \rho) \\
&\quad \mathit{fun}\ \rho\ (\mathit{Ref}\ \ v) = \mathit{readIOAddr}\ (\rho\ !\ v) \\
&\quad \mathit{arg}\ \rho\ (\mathit{Lam}\ l) = \mathit{return}\ \$\ \mathit{Clo}\ (l, \rho) \\
&\quad \mathit{arg}\ \rho\ (\mathit{Ref}\ \ v) = \mathit{readIOAddr}\ (\rho\ !\ v) \\
&\quad \mathit{addr} \mapsto v \qquad\quad\ \ = \mathit{writeIOAddr}\ \ \mathit{addr}\ \ v \\
&\quad \mathit{alloc}\ v \qquad\qquad\ \ = \mathit{liftM}\ \ \mathit{IOAddr}\ \$\ \mathit{newIORef}\ \bot \\
&\quad \mathit{tick}\ \_\ \_ \qquad\qquad\ = \mathit{return}\ ()
\end{aligned}
$$

Driver Loop

$$
\begin{aligned}
&interpret :: CExp \to IO\ (P\Sigma\ IOAddr) \\
&interpret\ e = go\ (e, Map.empty) \\
&\quad \textbf{where}\ go :: (P\Sigma\ IOAddr) \to IO\ (P\Sigma\ IOAddr) \\
&\qquad\qquad go\ s = \textbf{do}\ s' \leftarrow mnext\ s \\
&\qquad\qquad\qquad\qquad \textbf{case}\ s'\ \textbf{of}\ \begin{array}{ll} x@(Exit, \_) & \to return\ x \\ y & \to go\ y \end{array}
\end{aligned}
$$

Driver Loop

$$
\begin{aligned}
&interpret :: CExp \rightarrow IO\ (P\Sigma\ IOAddr) \\
&interpret\ e = go\ \boxed{(e, Map.empty)}\ s_0 \\
&\quad \textbf{where}\ go :: (P\Sigma\ IOAddr) \rightarrow IO\ (P\Sigma\ IOAddr) \\
&\qquad\quad go\ s = \textbf{do}\ s' \leftarrow mnext\ s \\
&\qquad\qquad\qquad \textbf{case}\ s'\ \textbf{of}\ x@(Exit, \_) \rightarrow return\ x \\
&\qquad\qquad\qquad\qquad\qquad\ y \qquad\qquad\quad \rightarrow go\ y
\end{aligned}
$$

Driver Loop

$$
\begin{aligned}
&interpret :: CExp \to IO\ (P\Sigma\ IOAddr)\\
&interpret\ e = go\ (e, Map.empty)\\
&\quad \textbf{where}\ go :: (P\Sigma\ IOAddr) \to IO\ (P\Sigma\ IOAddr)\\
&\qquad\qquad go\ s = \textbf{do}\ s' \leftarrow mnext\ s\\
&\qquad\qquad\qquad\qquad \textbf{case}\ s'\ \textbf{of}\ x@(Exit, \_) \to return\ x\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad y \qquad\qquad\quad \to go\ y
\end{aligned}
$$

s_0

Driver Loop

$$
\begin{aligned}
&\mathit{interpret} :: \mathit{CExp} \to \mathit{IO}\ (P\Sigma\ \mathit{IOAddr})\\
&\mathit{interpret}\ e = \mathit{go}\ \boxed{(e, \mathit{Map.empty})}\\
&\quad \textbf{where}\ \mathit{go} :: (P\Sigma\ \mathit{IOAddr}) \to \mathit{IO}\ (P\Sigma\ \mathit{IOAddr})\\
&\qquad\quad \mathit{go}\ s = \textbf{do}\ s' \leftarrow \boxed{\mathit{mnext}}\ s\\
&\qquad\qquad\qquad\quad \textbf{case}\ s'\ \textbf{of}\ x@(\mathit{Exit}, \_) \to \mathit{return}\ x\\
&\qquad\qquad\qquad\qquad\qquad\quad y \qquad\qquad\quad\ \to \mathit{go}\ y
\end{aligned}
$$

$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \ldots$

Instance II: Collecting Abstract Interpreter

Instance II: Collecting Abstract Interpreter

`State(State(List)) Monad`

+

Semantic Interface Implementation

+

Generic fixed point machinery

Collecting Semantics and Fixed Points

$$\hat{f} \in \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)$$

$$\hat{f}(\hat{S}) = \{\hat{\varsigma}_0\} \cup \{\hat{\varsigma}' \mid \hat{\varsigma} \leadsto \hat{\varsigma}' \text{ and } \hat{\varsigma} \in \hat{S}\}$$

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

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

$$\begin{array}{l}
\mathit{kleeneIt} :: (\mathit{Lattice}\ a) \Rightarrow (a \to a) \to a \\
\mathit{kleeneIt}\ f = \mathit{loop}\ \bot \\
\quad \textbf{where}\ \mathit{loop}\ c = \textbf{let}\ c' = f\ c\ \textbf{in} \\
\qquad\qquad\qquad\qquad \textbf{if}\ c' \sqsubseteq c\ \textbf{then}\ c\ \textbf{else}\ \mathit{loop}\ c'
\end{array}$$

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

$$\begin{aligned}
&\mathit{kleeneIt} :: (\mathit{Lattice}\ a) \Rightarrow (a \to a) \to a \\
&\mathit{kleeneIt}\ f = \mathit{loop}\ \bot \\
&\quad \mathbf{where}\ \mathit{loop}\ c = \mathbf{let}\ c' = f\ c\ \mathbf{in} \\
&\qquad\qquad\qquad\qquad \mathbf{if}\ c' \sqsubseteq c\ \mathbf{then}\ c\ \mathbf{else}\ \mathit{loop}\ c'
\end{aligned}$$

$$\begin{aligned}
&\mathbf{class}\ \mathit{Collecting}\ m\ a\ \mathit{fp} \mid \mathit{fp} \to a, \mathit{fp} \to m\ \mathbf{where} \\
&\quad \mathit{applyStep} :: (a \to m\ a) \to \mathit{fp} \to \mathit{fp} \\
&\quad \mathit{inject} :: a \to \mathit{fp}
\end{aligned}$$

$$\begin{aligned}
&\mathit{exploreFP} :: (\mathit{Lattice}\ \mathit{fp}, \mathit{Collecting}\ m\ a\ \mathit{fp}) \Rightarrow \\
&\qquad\qquad\quad (a \to m\ a) \to a \to \mathit{fp} \\
&\mathit{exploreFP}\ \mathit{step}\ c = \mathit{kleeneIt}\ \mathcal{F} \\
&\quad \mathbf{where}\ \mathcal{F}\ s = \mathit{inject}\ c \sqcup \mathit{applyStep}\ \mathit{step}\ s
\end{aligned}$$

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

$$\begin{aligned}
&\mathit{kleeneIt} :: (\mathit{Lattice}\ a) \Rightarrow (a \to a) \to a\\
&\mathit{kleeneIt}\ f\ = \mathit{loop}\ \bot\\
&\quad \mathbf{where}\ \mathit{loop}\ c = \mathbf{let}\ c' = f\ c\ \mathbf{in}\\
&\qquad\qquad\qquad\quad \mathbf{if}\ c' \sqsubseteq c\ \mathbf{then}\ c\ \mathbf{else}\ \mathit{loop}\ c'
\end{aligned}$$

$(\rightsquigarrow)$ →

$$\begin{aligned}
&\mathbf{class}\ \mathit{Collecting}\ m\ a\ \mathit{fp} \mid \mathit{fp} \to a, \mathit{fp} \to m\ \mathbf{where}\\
&\quad \mathit{applyStep} :: (a \to m\ a) \to \mathit{fp} \to \mathit{fp}\\
&\quad \mathit{inject} :: a \to \mathit{fp}
\end{aligned}$$

$$\begin{aligned}
&\mathit{exploreFP} :: (\mathit{Lattice}\ \mathit{fp}, \mathit{Collecting}\ m\ a\ \mathit{fp}) \Rightarrow\\
&\qquad\qquad\quad (a \to m\ a) \to a \to \mathit{fp}\\
&\mathit{exploreFP}\ \mathit{step}\ c = \mathit{kleeneIt}\ \mathcal{F}\\
&\quad \mathbf{where}\ \mathcal{F}\ s = \mathit{inject}\ c \sqcup \mathit{applyStep}\ \mathit{step}\ s
\end{aligned}$$

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

$$\begin{array}{l}
\mathit{kleeneIt} :: (\mathit{Lattice}\ a) \Rightarrow (a \to a) \to a \\
\mathit{kleeneIt}\ f = \mathit{loop}\ \bot \\
\quad \textbf{where}\ \mathit{loop}\ c = \textbf{let}\ c' = f\ c\ \textbf{in} \\
\qquad\qquad\qquad\qquad \textbf{if}\ c' \sqsubseteq c\ \textbf{then}\ c\ \textbf{else}\ \mathit{loop}\ c'
\end{array}$$

$(\rightsquigarrow)$ → *applyStep*

$\{\cdot\}$ → *inject*

$$\begin{array}{l}
\textbf{class}\ \mathit{Collecting}\ m\ a\ \mathit{fp} \mid \mathit{fp} \to a, \mathit{fp} \to m\ \textbf{where} \\
\quad \mathit{applyStep} :: (a \to m\ a) \to \mathit{fp} \to \mathit{fp} \\
\quad \mathit{inject} :: a \to \mathit{fp}
\end{array}$$

$$\begin{array}{l}
\mathit{exploreFP} :: (\mathit{Lattice}\ \mathit{fp}, \mathit{Collecting}\ m\ a\ \mathit{fp}) \Rightarrow \\
\qquad\qquad\quad (a \to m\ a) \to a \to \mathit{fp} \\
\mathit{exploreFP}\ \mathit{step}\ c = \mathit{kleeneIt}\ \mathcal{F} \\
\quad \textbf{where}\ \mathcal{F}\ s = \mathit{inject}\ c \sqcup \mathit{applyStep}\ \mathit{step}\ s
\end{array}$$

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

$$
\begin{array}{l}
\mathit{kleeneIt} :: (\mathit{Lattice}\ a) \Rightarrow (a \to a) \to a \\
\mathit{kleeneIt}\ f = \mathit{loop}\ \bot \\
\quad \textbf{where}\ \mathit{loop}\ c = \textbf{let}\ c' = f\ c\ \textbf{in} \\
\qquad\qquad\qquad\qquad \textbf{if}\ c' \sqsubseteq c\ \textbf{then}\ c\ \textbf{else}\ \mathit{loop}\ c'
\end{array}
$$

$$
\begin{array}{l}
\textbf{class}\ \mathit{Collecting}\ m\ a\ \mathit{fp} \mid \mathit{fp} \to a, \mathit{fp} \to m\ \textbf{where} \\
\mathit{applyStep} :: (a \to m\ a) \to \mathit{fp} \to \mathit{fp} \\
\mathit{inject} :: a \to \mathit{fp}
\end{array}
$$

$(\leadsto)$ → $\mathit{applyStep}$

$\{\cdot\}$ → inject

$$
\begin{array}{l}
\mathit{exploreFP} :: (\mathit{Lattice}\ \mathit{fp}, \mathit{Collecting}\ m\ a\ \mathit{fp}) \Rightarrow \\
\qquad\qquad (a \to m\ a) \to a \to \mathit{fp} \\
\mathit{exploreFP}\ \mathit{step}\ c = \mathit{kleeneIt}\ \mathcal{F} \\
\quad \textbf{where}\ \mathcal{F}\ s = \underbrace{\mathit{inject}\ c \sqcup \mathit{applyStep}\ \mathit{step}\ s}_{\hat{f}(\hat{S}) = \{\hat{\varsigma}_0\} \cup \{\hat{\varsigma}' \mid \hat{\varsigma} \leadsto \hat{\varsigma}' \text{ and } \hat{\varsigma} \in \hat{S}\}}
\end{array}
$$

$$
\begin{aligned}
runAnalysis &:: (CPSInterface\ m\ a, Lattice\ fp,\\
&\qquad Collecting\ m\ (P\Sigma\ a)\ fp) \Rightarrow\\
&\qquad CExp \rightarrow fp\\
runAnalysis\ e &= exploreFP\ mnext\ (e, Map.empty)
\end{aligned}
$$

$$
\begin{aligned}
runAnalysis :: (&CPSInterface\ m\ a, Lattice\ fp,\\
&Collecting\ m\ (P\Sigma\ a)\ fp) \Rightarrow\\
&CExp \rightarrow fp
\end{aligned}
$$

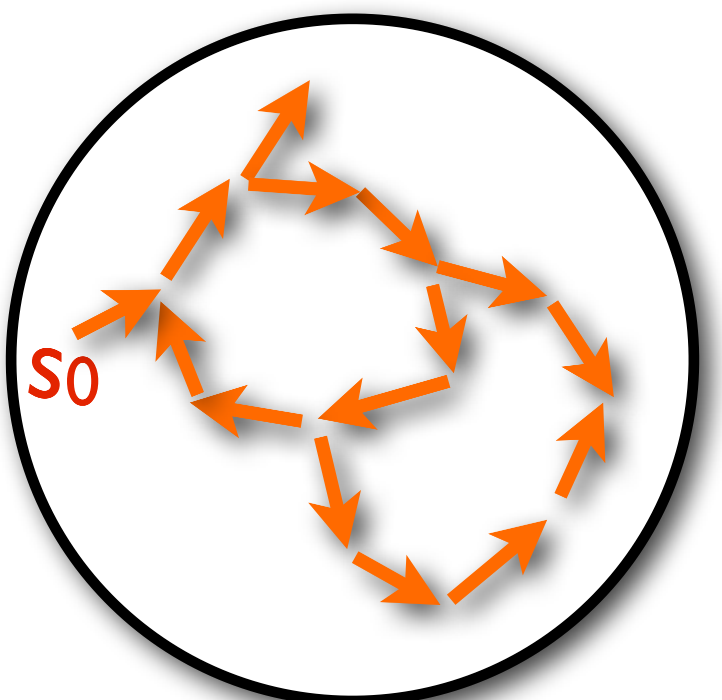
$$
runAnalysis\ e = exploreFP\ mnext\ \underbrace{\boxed{(e, Map.empty)}}_{s_0}
$$

$$runAnalysis :: (CPSInterface\ m\ a, Lattice\ fp,$$
$$Collecting\ m\ (P\Sigma\ a)\ fp) \Rightarrow$$
$$CExp \rightarrow fp$$
$$runAnalysis\ e = exploreFP\ mnext\ \boxed{(e, Map.empty)}$$

s_0

$$
\begin{aligned}
runAnalysis :: (&CPSInterface\ m\ a, Lattice\ fp,\\
&Collecting\ m\ (P\Sigma\ a)\ fp) \Rightarrow\\
&CExp \rightarrow fp
\end{aligned}
$$

$$runAnalysis\ e = exploreFP\ mnext\ \boxed{(e, Map.empty)}$$



Implementing
Collecting Abstract Interpreter
in 3 steps

1. Fixing the Monad

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

1. Fixing the Monad

$$\textbf{type}\ StorePassing\ s\ g = StateT\ g\ (StateT\ s\ \overbrace{[\,]}^{\text{non-determinism}})$$

1. Fixing the Monad

$$\mathbf{type}\ StorePassing\ s\ g = StateT\ g\ (\underbrace{StateT\ s}_{\text{store}}\ \overbrace{[\,]}^{\text{non-determinism}})$$

1. Fixing the Monad

$$\mathbf{type}\ StorePassing\ s\ g = \underbrace{StateT\ g}_{\text{time}}\ (\underbrace{StateT\ s}_{\text{store}}\ \overbrace{[\,]}^{\text{non-determinism}})$$

2. Providing Denotations

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

3. Starting and Stepping

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

3. Starting and Stepping

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

starting

3. Starting and Stepping

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

starting

stepping

$$runAnalysis\ exp :: \mathcal{P}\ ((P\Sigma\ Integer, Integer), Store\ Integer)$$

a program

$$runAnalysis\ \boxed{exp} :: \mathcal{P}\ ((P\Sigma\ Integer, Integer), Store\ Integer)$$

a program

$$runAnalysis\ \boxed{exp} :: \mathcal{P}\ ((P\Sigma\ Integer, Integer), Store\ Integer)$$

$$\underbrace{\qquad\qquad\qquad\qquad}_{(P\Sigma\ Addr) \times Time \times (Store\ Addr)}$$

$$\text{where } Addr = Integer$$

$$Time = Integer$$

Abstract Interpretation
can be seen as a computational effect

Monadic refactoring disentangles
transitions from their denotation

A monad specifies the state-space

Check the paper

- Generic implementation of polyvariance
- Language-independent store
- Language-independent abstract counting
- Reusable abstract garbage collection
- Pluggable widening strategies

Try the code

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

- Featherweight Java
- Direct-style λ-calculus
- Monadic machinery
- Full-fledged abstract GC
- Counting
- Lots of examples

Try the code

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

- Featherweight Java
- Direct-style λ-calculus
- Monadic machinery
- Full-fledged abstract GC
- Counting
- Lots of examples

Thanks