Modular Analysis

Joonhyup Lee

May 17, 2024

Abstract

We present a framework of modular static analysis that analyzes open program fragments in advance and complete the whole analysis later when the fragments are closed. The framework is defined for a call-by-value lambda calculus extended with constructs for defining and linking first-class modules (a collection of bindings) that support recursive bindings for values, modules, and functors(module functions). Thanks to the abstract interpretation framework, our modular analysis framework's key is how to define the semantics, called "shadow" semantics of open program fragments so that the computation with free variables should be captured in advance and be completed later at link-time. A modular static analysis is abstractions of this shadow semantics and of the linking operation. The safety of the framework is proven in Coq. Two instances of the framwork are presented: value analyses for an applicative language(control-flow, closure analysis) and for an imperative language.

1 Problem

By modular analysis we mean a static analysis technology that analyzes programs in the compositional way. Modular analysis partially analyzes program fragments separately and then complete the analysis when all the fragments are available and linked.

Designing a modular analysis decides on two factors: how to build sound partial semantic summaries of program fragments and how to complete them at link time. A particular modular analysis strikes a balance between the two factors. Depending on how much analysis work is invested on each factor, modular analyses range from one extreme, a trivial one (wait until the whole code is available and do the whole-program analysis) to another extreme, non-trivial yet hardly-automatic ones (abduct assumptions about free variables and check them at link-time).

However, designing a modular analysis has been elusive. There is a lack of a general framework by which we can design sound modular analyses with varying balance and accuracy of our choice. We need a general framework by which static analysis designers can design non-trivial sound modular analyses.

2 Framework Sketch

Thanks to the abstract interpretation framework, the key in our modular analysis framework is how to define the semantics (we call it "shadow" semantics) of open program fragments so that the computation with free variables should be captured in advance and be completed later at link-time. A modular static analysis is then nothing but abstractions of this shadow semantics and of the linking operation.

2.1 Examples of Shadow Semantics and its Abstraction

Shadow semantics is the semantics for the computation involving free variables. We give examples of shadows for program fragments written in OCaml.

Example 1: Mapping an Unknown Function Over a Known List

```
let rec map = fun f -> fun l ->
  match [0]l with
  | [] -> []
  | hd :: tl -> [1]([3]f [4]hd) :: [2](map f tl)
in map [5]g [6](1 :: 2 :: 3 :: [])
```

The shadow semantics of the above fragment is

$$\{\mathsf{Call}(\mathrm{G}, 1) :: \mathsf{Call}(\mathrm{G}, 2) :: \mathsf{Call}(\mathrm{G}, 3) :: []\} \text{ where } \mathrm{G} = \mathsf{Read}(\mathsf{Init}, \mathsf{g})$$

Shadows such as Call, Read correspond to semantic operations, like function application and reading from the environment. The Init shadow is the unknown initial environment that the fragment will execute under.

Computing a sound and finite approximation of the shadow semantics of program fragments corresponds to building a sound partial semantic summaries in advance. For the above example, we may use an abstraction that stores summaries of each input *environment* and output *value* per program point.

Notation. We write p_i for each program point i. p_i.in denotes the input to p_i, and p_i.out denotes the output from p_i. For brevity, we omit the selector in p_i.out and write p_i. We write $[a, b]_{\mathrm{itv}}$ to denote the interval from integer a to b.

Now, the abstract shadow

$$\{p_1 :: p_2\}$$

is a sound approximation of the concrete shadow. To illustrate why, here are abstract shadows from some program points involved in the map computation.

p_1.out : $\{\mathsf{Call}^{\#}(p_3, p_4)\}$ $\quad$ p_4.out : $[1, 3]_{\mathrm{itv}}$

p_2.out : $\{[], p_1 :: p_2\}$ $\quad$ p_5.in : $\{\mathsf{Init}^{\#}, \mathtt{map} \mapsto \{\langle \lambda \mathtt{f} \ldots \rangle\}\}$

p_3.out : $\{\mathsf{Read}^{\#}(p_5.\mathrm{in}, \mathsf{g})\}$

Starting from $p_1 :: p_2$, we can reach the concrete shadow by expanding each program point according to the above table.

Example 2: Mapping an Unknown Function Over an Unknown List

```
let rec map = ... in map 5g 6lst
```

A more interesting case is when even the list is replaced with a free variable. The shadow semantics of the above fragment is

$$\{[], \mathsf{Call}(\mathrm{G}, \mathrm{Hd}(\mathrm{Lst})) :: [], \mathsf{Call}(\mathrm{G}, \mathrm{Hd}(\mathrm{Tl}(\mathrm{Lst}))) :: \mathsf{Call}(\mathrm{G}, \mathrm{Hd}(\mathrm{Lst})) :: [],$$
$$\ldots, \mathsf{Call}(\mathrm{G}, \mathrm{Hd}(\mathrm{Tl}^n(\mathrm{Lst}))) :: \cdots :: [], \ldots\}$$

where

$$\mathrm{G} = \mathsf{Read}(\mathsf{Init}, \mathtt{g}) \qquad \mathrm{Lst} = \mathsf{Read}(\mathsf{Init}, \mathtt{lst})$$
$$\mathrm{Hd}(l) = \mathsf{Match}(l, ::, 0) \qquad \mathrm{Tl}(l) = \mathsf{Match}(l, ::, 1)$$

The same abstraction as in Example 1 results in the same abstract shadow

$$\{p_1 :: p_2\}$$

but with different shadows for each program point.

$$p_0.\mathrm{out} : \{\mathsf{Read}^{\#}(p_6.\mathrm{in}, \mathtt{lst}), \mathsf{Match}^{\#}(p_0, ::, 1)\} \qquad p_3.\mathrm{out} : \{\mathsf{Read}^{\#}(p_5.\mathrm{in}, \mathtt{g})\}$$
$$p_1.\mathrm{out} : \{\mathsf{Call}^{\#}(p_3, p_4)\} \qquad p_4.\mathrm{out} : \{\mathsf{Match}^{\#}(p_0, ::, 0)\}$$
$$p_2.\mathrm{out} : \{[], p_1 :: p_2\} \qquad p_5.\mathrm{in} : \{\mathsf{Init}^{\#}, \mathtt{map} \mapsto \{\langle \lambda \mathtt{f} \ldots \rangle\}\}$$
$$p_6.\mathrm{in} : \{\mathsf{Init}^{\#}, \mathtt{map} \mapsto \{\langle \lambda \mathtt{f} \ldots \rangle\}\}$$

Example 3: An Imperative Swap

```
let t = !x in1 x := !y;2 y := temp3
```

The shadow semantics can be extended to support imperative features. We simply have to add shadows that correspond to *dereferencing*, a semantic operation central to imperative programs.

For this example, the environment and memory at each program point is

	Environment	Memory
p_1	$\langle\{\mathtt{t} \mapsto *\mathrm{X}\}, \mathsf{Init}\rangle$	$\langle\{\{\mathrm{X}\} \mapsto *\mathrm{X}\}, \mathsf{Init}\rangle$
p_2	$\langle\{\mathtt{t} \mapsto *\mathrm{X}\}, \mathsf{Init}\rangle$	$\langle\{\{\mathrm{X}, \mathrm{Y}\} \mapsto *\mathrm{X}\}, \mathsf{Init}\rangle$ or $\langle\{\{\mathrm{X}\} \mapsto *\mathrm{Y}, \{\mathrm{Y}\} \mapsto *\mathrm{Y}\}, \mathsf{Init}\rangle$
p_3	$\langle\{\mathtt{t} \mapsto *\mathrm{X}\}, \mathsf{Init}\rangle$	$\langle\{\{\mathrm{X}, \mathrm{Y}\} \mapsto *\mathrm{X}\}, \mathsf{Init}\rangle$ or $\langle\{\{\mathrm{X}\} \mapsto *\mathrm{Y}, \{\mathrm{Y}\} \mapsto *\mathrm{X}\}, \mathsf{Init}\rangle$

where

$$\mathrm{X} = \mathsf{Read}(\mathsf{Init}, \mathtt{x}) \quad \mathrm{Y} = \mathsf{Read}(\mathsf{Init}, \mathtt{y}) \quad *S = \mathsf{DerefShadow}(\mathsf{Init}, S) \text{ for shadow } S$$

The memory is represented by a tuple holding the bindings for each concrete address and the most recent shadow that affected the memory. The concrete address consists of addresses that are allocated during the execution of the program, and the *alias set* of shadows. The alias set is updated each time there

is a read or a write to a shadow address, and the memory retains the invariant that the alias sets must be mutually disjoint. In the above example, the alias set is updated when the memory is dereferenced with X and Y. The resulting memory depends on aliasing, as expected.

The "most recent shadow" must be recorded due to unresolved function calls. For example, if there is a call to a *shadow* G with argument v under memory

$$m = \langle \{(a_1, \{S_1, S_2\}) \mapsto v_1, \{S_3\} \mapsto v_2\}, S_4 \rangle$$

the memory will be updated to

$$\langle \{(a_1, \{S_1, S_2\}) \mapsto \mathsf{DerefAddr}(\mathrm{C}, a_1), \ldots, \{S_3\} \mapsto \mathsf{DerefShadow}(\mathrm{C}, S_3)\}, \mathrm{C} \rangle$$

where

$$\mathrm{C} = \mathsf{Call}(\mathrm{G}, v|m) \quad (m \text{ records that the call happened under memory } m)$$

For efficient abstraction of the shadow semantics, one may employ widening. For instance, each shadow in the alias sets may be kept in its concrete form up until some threshold. After that, the memory may be collapsed to prevent blowup.

2.2 Examples of Linking and its Abstraction

The shadow semantics become actual when the involved free variables are known at link time. The linking semantics, the semantics of the link operation, defines this actualization operation. For the above fragments, let's consider that a closing fragment is available.

Example 4: Module Exporting a Known Function

```
let g = 7fun x -> x + 1
```

This is a module which returns the environment $\sigma = \{\mathtt{g} \mapsto \langle \lambda\mathtt{x.x + 1}, [] \rangle\}$. Linking this with the concrete shadow in Example 1 gives:

$$\{2 :: 3 :: 4 :: []\}$$

Computing a sound and finite approximation of the linking semantics corresponds to completing partial analysis summaries at link time when the analysis results for the involved free variable are available. For the above example,

$$\sigma^{\#} = \{\mathtt{g} \mapsto \{\langle \lambda\mathtt{x.x + 1}, p_7.\mathrm{in} \rangle\}\} \text{ where } p_7.\mathrm{in} : []$$

is the abstract shadow.

Applying a sound abstract version of the linking operator will result in

$$p_1.\mathrm{out} : [2, 4]_{\mathrm{itv}} \qquad p_4.\mathrm{out} : [1, 3]_{\mathrm{itv}}$$
$$p_2.\mathrm{out} : \{[], p_1 :: p_2\} \qquad p_5.\mathrm{in} : \sigma^{\#}$$
$$p_3.\mathrm{out} : \{\langle \lambda\mathtt{x.x + 1}, p_7.\mathrm{in} \rangle\}$$

Example 5: Module Exporting a Foreign Function

```
external g : int -> int = "incr"
```

The function g might also be a foreign function. The return value of this module is $\sigma = \{\mathsf{g} \mapsto \mathsf{Prim}(\texttt{incr})\}$, where Prim stands for a *primitive* value. Linking this with the concrete shadow in Example 1 gives:

$$\{\mathsf{PrimCall}(\texttt{incr}, 1) :: \mathsf{PrimCall}(\texttt{incr}, 2) :: \mathsf{PrimCall}(\texttt{incr}, 3) :: []\}$$

The sound abstraction for σ is

$$\sigma^{\#} = \{\mathsf{g} \mapsto \{\mathsf{Prim}(\texttt{incr})\}\}$$

Applying a sound abstract version of the linking operator will result in

$$p_1.\text{out} : \{\mathsf{PrimCall}^{\#}(\texttt{incr}, p_4)\} \qquad p_4.\text{out} : [1, 3]_{\text{itv}}$$
$$p_2.\text{out} : \{[], p_1 :: p_2\} \qquad p_5.\text{in} : \sigma^{\#}$$
$$p_3.\text{out} : \{\mathsf{Prim}(\texttt{incr})\}$$

Example 6: Partial Resolution

```
let lst = 7 1 :: 8(9 2 :: 10(11 3 :: 12[]))
```

The return value of this module is $\sigma = \langle\{\texttt{lst} \mapsto 1 :: 2 :: 3 :: []\}, \mathsf{Init}\rangle$. Linking this with the concrete shadow in Example 2 gives:

$$\{[], \mathsf{Call}(\mathrm{G}, 1) :: [], \mathsf{Call}(\mathrm{G}, 1) :: \mathsf{Call}(\mathrm{G}, 2) :: [], \mathsf{Call}(\mathrm{G}, 1) :: \mathsf{Call}(\mathrm{G}, 2) :: \mathsf{Call}(\mathrm{G}, 3) :: []\}$$

where

$$\mathrm{G} = \mathsf{Read}(\mathsf{Init}, \mathsf{g})$$

The reason for this imprecision is because the concrete shadows dropped some constraints. For example, the output of Example 2 can be [] if and only if Tl(Lst) is matched with []. To prevent this, shadows may be augmented with constraints, such as $([], \mathrm{Tl}(\mathrm{Lst}) \doteq [])$, but we elide this detail for presentation.

The sound abstraction for σ is

$\sigma^{\#} = \{\mathsf{Init}^{\#}, \texttt{lst} \mapsto \{p_7 :: p_8\}\}$ where $p_7.\text{out} : [1, 1]_{\text{itv}}$, $p_8.\text{out} : \{p_9 :: p_{10}\} \ldots$

The result of linking $\sigma^{\#}$ to the abstract shadow in Example 2 is equivalent to the result of Example 1. This means that the shadows can be filled in *incrementally*.

2.3 Paper Overview

We present our framework for a call-by-value lambda calculus extended with constructs for defining and linking first-class modules (a collection of bindings) that support recursive bindings for values, modules, and functors(module functions).

The framework shows two points: how to define the shadow semantics and what to prove for the soundness of consequent modular analysis. The safety of the framework is proven in Coq. We present two instances of the framework: for high-order applicative language we show modular closure analysis design, and for imperative languages we show modular [TODO] analysis design.

3 Syntax and Semantics

3.1 Abstract Syntax

Identifiers	x	$\in$	Var	
Expression	e	$\to$	$x \mid \lambda x.e \mid e\ e$	λ-calculus
		$\mid$	$e \rtimes e$	linked expression
		$\mid$	ε	empty module
		$\mid$	$x = e\ ;\ e$	(recursive) binding

Figure 1: Abstract syntax of the language.

3.2 Operational Semantics

Environment	σ	$\in$	$\mathrm{Env} \triangleq \{\bullet\} + \mathrm{Var} \times (\mathrm{Loc} + \mathrm{WVal}) \times \mathrm{Env}$	
Location	ℓ	$\in$	Loc	
Value	v	$\in$	$\mathrm{Val} \triangleq \mathrm{Env} + \mathrm{Var} \times \mathrm{Expr} \times \mathrm{Env}$	
Weak Value	w	$\in$	$\mathrm{WVal} \triangleq \mathrm{Val} + \underline{\mathrm{Val}}$	
Environment	σ	$\to$	$\bullet$	empty stack
		$\mid$	$(x, w) :: \sigma$	weak value binding
		$\mid$	$(x, \ell) :: \sigma$	free location binding
Value	v	$\to$	σ	exported environment
		$\mid$	$\langle \lambda x.e, \sigma \rangle$	closure
Weak Value	w	$\to$	v	value
		$\mid$	$\mu\ell.v$	recursive value

Figure 2: Definition of the semantic domains.

The big-step operational semantics is *deterministic* up to α-equivalence. We have also proven the bisimilarity of this semantics with a more conventional one that uses backpatching for arbitrary recursive definitions.

$\boxed{\sigma \vdash e \Downarrow v}$

$$\textsc{Id}\ \frac{\sigma(x) = v}{\sigma \vdash x \Downarrow v} \qquad \textsc{RecId}\ \frac{\sigma(x) = \mu\ell.v}{\sigma \vdash x \Downarrow v[\mu\ell.v/\ell]} \qquad \textsc{Fn}\ \frac{}{\sigma \vdash \lambda x.e \Downarrow \langle \lambda x.e, \sigma \rangle}$$

$$\textsc{App}\ \frac{\sigma \vdash e_1 \Downarrow \langle \lambda x.e, \sigma_1 \rangle \qquad \sigma \vdash e_2 \Downarrow v_2 \qquad (x, v_2) :: \sigma_1 \vdash e \Downarrow v}{\sigma \vdash e_1\ e_2 \Downarrow v}$$

$$\textsc{Link}\ \frac{\sigma \vdash e_1 \Downarrow \sigma_1 \qquad \sigma_1 \vdash e_2 \Downarrow v}{\sigma \vdash e_1 \rtimes e_2 \Downarrow v} \qquad \textsc{Empty}\ \frac{}{\sigma \vdash \varepsilon \Downarrow \bullet}$$

$$\textsc{Bind}\ \frac{\ell \notin \mathrm{FLoc}(\sigma) \qquad (x, \ell) :: \sigma \vdash e_1 \Downarrow v_1 \qquad (x, \mu\ell.v_1) :: \sigma \vdash e_1 \Downarrow \sigma_2}{\sigma \vdash x = e_1; e_2 \Downarrow (x, \mu\ell.v_1) :: \sigma_2}$$

Figure 3: The big-step operational semantics.

4 Generating and Resolving Shadows

Now we formulate the semantics for generating shadows.

Shadow	S	$\rightarrow$	Init	initial environment
		$\mid$	$\mathsf{Read}(S, x)$	read shadow
		$\mid$	$\mathsf{Call}(S, v)$	call shadow
Environment	σ	$\rightarrow$	$\cdots$	
		$\mid$	$[S]$	answer to an shadow
Value	v	$\rightarrow$	$\cdots$	
		$\mid$	S	answer to an shadow

Figure 4: Definition of the semantic domains with shadows. All other semantic domains are equal to Figure 2.

We extend how to read weak values given an environment.

$$\bullet(x) \triangleq \bot \qquad ((x', \ell) :: \sigma)(x) \triangleq (x = x' \ ?\ \ell : \sigma(x))$$
$$[S](x) \triangleq \mathsf{Read}(S, x) \qquad ((x', w) :: \sigma)(x) \triangleq (x = x' \ ?\ w : \sigma(x))$$

Then we need to add only three rules to the semantics in Figure 3 for the semantics to incorporate shadows.

$$\textsc{LinkShadow}\ \frac{\sigma \vdash e_1 \Downarrow S \qquad [S] \vdash e_2 \Downarrow v}{\sigma \vdash e_1 \rtimes e_2 \Downarrow v} \qquad \textsc{AppShadow}\ \frac{\sigma \vdash e_1 \Downarrow S \qquad \sigma \vdash e_2 \Downarrow v}{\sigma \vdash e_1\ e_2 \Downarrow \mathsf{Call}(S, v)}$$

$$\textsc{BindShadow}\ \frac{\ell \notin \mathrm{FLoc}(\sigma) \qquad (x, \ell) :: \sigma \vdash e_1 \Downarrow v_1 \qquad (x, \mu\ell.v_1) :: \sigma \vdash e_1 \Downarrow S_2}{\sigma \vdash x = e_1; e_2 \Downarrow (x, \mu\ell.v_1) :: [S_2]}$$

Now we need to formulate the *concrete linking* rules. The concrete linking rule $\sigma_0 \rtimes w$, given an answer σ_0 to the Init shadow, resolves all shadows within w to obtain a set of final results.

$$\boxed{\rtimes \in \mathrm{Env} \to \mathrm{Shadow} \to \mathcal{P}(\mathrm{Val})}$$

$$\begin{aligned}
\sigma_0 \rtimes \mathsf{Init} &\triangleq \{\sigma_0\} \\
\sigma_0 \rtimes \mathsf{Read}(S, x) &\triangleq \{v_+ | \sigma_+ \in \sigma_0 \rtimes S, \sigma_+(x) = v_+\} \\
&\quad \cup \{v_+[\mu\ell.v_+/\ell] | \sigma_+ \in \sigma_0 \rtimes E, \sigma_+(x) = \mu\ell.v_+\} \\
\sigma_0 \rtimes \mathsf{Call}(S, v) &\triangleq \{v'_+ | \langle \lambda x.e, \sigma_+ \rangle \in \sigma_0 \rtimes E, v_+ \in \sigma_0 \rtimes v, (x, v_+) :: \sigma_+ \vdash e \Downarrow v'_+\} \\
&\quad \cup \{\mathsf{Call}(S_+, v_+) | S_+ \in \sigma_0 \rtimes E, v_+ \in \sigma_0 \rtimes v\}
\end{aligned}$$

$$\boxed{\rtimes \in \mathrm{Env} \to \mathrm{Env} \to \mathcal{P}(\mathrm{Env})}$$

$$\begin{aligned}
\sigma_0 \rtimes \bullet &\triangleq \{\bullet\} \\
\sigma_0 \rtimes (x, \ell) :: \sigma &\triangleq \{(x, \ell) :: \sigma_+ | \sigma_+ \in \sigma_0 \rtimes \sigma\} \\
\sigma_0 \rtimes (x, w) :: \sigma &\triangleq \{(x, w_+) :: \sigma_+ | w_+ \in \sigma_0 \rtimes w, \sigma_+ \in \sigma_0 \rtimes \sigma\} \\
\sigma_0 \rtimes [S] &\triangleq \{\sigma_+ | \sigma_+ \in \sigma_0 \rtimes S\} \cup \{[S_+] | S_+ \in \sigma_0 \rtimes S\}
\end{aligned}$$

$$\boxed{\rtimes \in \mathrm{Env} \to \mathrm{Val} \to \mathcal{P}(\mathrm{Val})}$$

$$\sigma_0 \rtimes \langle \lambda x.e, \sigma \rangle \triangleq \{\langle \lambda x.e, \sigma_+ \rangle | \sigma_+ \in \sigma_0 \rtimes \sigma\}$$

$$\boxed{\rtimes \in \mathrm{Env} \to \mathrm{WVal} \to \mathcal{P}(\mathrm{WVal})}$$

$$\sigma_0 \rtimes \mu\ell.v \triangleq \{\mu\ell'.v_+ | \ell' \notin \mathrm{FLoc}(v) \cup \mathrm{FLoc}(\sigma_0), v_+ \in \sigma_0 \rtimes v[\ell'/\ell]\}$$

Concrete linking makes sense because of the following theorem. First define:

$$\mathrm{eval}(e, \sigma) \triangleq \{v | \sigma \vdash e \Downarrow v\} \quad \mathrm{eval}(e, \Sigma) \triangleq \bigcup_{\sigma \in \Sigma} \mathrm{eval}(e, \sigma) \quad \Sigma_0 \rtimes W \triangleq \bigcup_{\substack{\sigma_0 \in \Sigma_0 \\ w \in W}} (\sigma_0 \rtimes w)$$

Then the following holds:

Theorem 4.1 (Advance)**.** *Given* $e \in \mathrm{Expr}, \Sigma_0, \Sigma \subseteq \mathrm{Env}$*,*

$$\mathrm{eval}(e, \Sigma_0 \rtimes \Sigma) \subseteq \Sigma_0 \rtimes \mathrm{eval}(e, \Sigma)$$

Now we can formulate modular analysis. A modular analysis consists of two requirements: an abstraction for the semantics with shadows and an abstraction for the semantic linking operator.

Theorem 4.2 (Modular analysis)**.** *Assume:*

1. *An abstract domain* $\mathrm{WVal}^\#$ *and a monotonic* $\gamma \in \mathrm{WVal}^\# \to \mathcal{P}(\mathrm{WVal})$
2. *A sound* $\mathrm{eval}^\#$*:* $\Sigma_0 \subseteq \gamma(\sigma_0^\#) \Rightarrow \mathrm{eval}(e, \Sigma_0) \subseteq \gamma(\mathrm{eval}^\#(e, \sigma_0^\#))$
3. *A sound* $\rtimes^\#$*:* $\Sigma_0 \subseteq \gamma(\sigma_0^\#)$ and $W \subseteq \gamma(w^\#) \Rightarrow \Sigma_0 \rtimes W \subseteq \gamma(\sigma_0^\# \rtimes^\# w^\#)$

then we have:

$$\Sigma_0 \subseteq \gamma(\sigma_0^\#) \text{ and } \Sigma \subseteq \gamma(\sigma^\#) \Rightarrow \mathrm{eval}(e, \Sigma_0 \rtimes \Sigma) \subseteq \gamma(\sigma_0^\# \rtimes^\# \mathrm{eval}^\#(e, \sigma^\#))$$

Corollary 4.1 (Modular analysis of linked program)**.**

$$\Sigma_0 \subseteq \gamma(\sigma_0^{\#}) \text{ and } [\mathsf{Init}] \in \gamma(\mathsf{Init}^{\#}) \Rightarrow$$
$$\mathrm{eval}(e_1 \rtimes e_2, \Sigma_0) \subseteq \gamma(\mathrm{eval}^{\#}(e_1, \sigma_0^{\#}) \rtimes^{\#} \mathrm{eval}^{\#}(e_2, \mathsf{Init}^{\#}))$$

5 CFA

5.1 Collecting semantics

$$
\begin{array}{rrcl}
\text{Program point} & p & \in & \mathbb{P} \triangleq \{\text{finite set of program points}\} \\
\text{Labelled expression} & pe & \in & \mathbb{P} \times \text{Expr} \\
\text{Labelled location} & \ell^p & \in & \mathbb{P} \times \text{Loc} \\
\text{Collecting semantics} & t & \in & \mathbb{T} \triangleq \mathbb{P} \to \mathcal{P}(\text{Env} + \text{Env} \times \text{Val}) \\
\text{Labelled expression} & pe & \to & \{p : e\} \\
\text{Expression} & e & \to & x \mid \lambda x.pe \mid pe\ pe \mid pe \rtimes pe \mid \varepsilon \mid x = pe; pe
\end{array}
$$

$\boxed{\text{Step} : \mathbb{T} \to \mathbb{T}}$

$$\text{Step}(t) \triangleq \bigcup_{p \in \mathbb{P}} \text{step}(t, p)$$

$\boxed{\text{step} : (\mathbb{T} \times \mathbb{P}) \to \mathbb{T}}$

$$\text{step}(t,p) \triangleq [p \mapsto \{(\sigma, v) | \sigma \in t(p) \text{ and } \sigma(x) = v\}] \qquad \text{when } \{p : x\}$$
$$\cup\, [p \mapsto \{(\sigma, v[\mu\ell^{p'}.v/\ell^{p'}]) | \sigma \in t(p) \text{ and } \sigma(x) = \mu\ell^{p'}.v\}]$$

$$\text{step}(t,p) \triangleq [p \mapsto \{(\sigma, \langle \lambda x.p', \sigma \rangle) | \sigma \in t(p)\}] \qquad \text{when } \{p : \lambda x.p'\}$$

$$\text{step}(t,p) \triangleq [p_1 \mapsto \{\sigma \in \text{Env} | \sigma \in t(p)\}] \qquad \text{when } \{p : p_1\ p_2\}$$
$$\cup\, [p_2 \mapsto \{\sigma \in \text{Env} | \sigma \in t(p)\}]$$
$$\cup \bigcup_{\sigma \in t(p)} \bigcup_{(\sigma, \langle \lambda x.p', \sigma_1 \rangle) \in t(p_1)} [p' \mapsto \{(x, v_2) :: \sigma_1 | (\sigma, v_2) \in t(p_2)\}]$$
$$\cup\, [p \mapsto \bigcup_{\sigma \in t(p)} \bigcup_{(\sigma, \langle \lambda x.p', \sigma_1 \rangle) \in t(p_1)} \bigcup_{(\sigma, v_2) \in t(p_2)} \{(\sigma, v) | ((x, v_2) :: \sigma_1, v) \in t(p')\}]$$
$$\cup\, [p \mapsto \bigcup_{\sigma \in t(p)} \{(\sigma, \mathsf{Call}(S_1, v_2)) | (\sigma, S_1) \in t(p_1) \text{ and } (\sigma, v_2) \in t(p_2)\}]$$

$$\text{step}(t,p) \triangleq [p_1 \mapsto \{\sigma | \sigma \in t(p)\}] \qquad \text{when } \{p : p_1 \rtimes p_2\}$$
$$\cup\, [p_2 \mapsto \bigcup_{\sigma \in t(p)} \{\sigma_1 | (\sigma, \sigma_1) \in t(p_1)\}]$$
$$\cup\, [p \mapsto \bigcup_{\sigma \in t(p)} \bigcup_{(\sigma, \sigma_1) \in t(p_1)} \{(\sigma, v_2) | (\sigma_1, v_2) \in t(p_2)\}]$$

$$\text{step}(t,p) \triangleq [p \mapsto \{(\sigma, \bullet) | \sigma \in t(p)\}] \qquad \text{when } \{p : \varepsilon\}$$

$$\text{step}(t,p) \triangleq [p_1 \mapsto \bigcup_{\sigma \in t(p)} \{(x, \ell^{p_1}) :: \sigma | \ell \notin \text{FLoc}(\sigma)\}] \qquad \text{when } \{p : x = p_1; p_2\}$$
$$\cup\, [p_2 \mapsto \bigcup_{\sigma \in t(p)} \{(x, \mu\ell^{p_1}.v_1) :: \sigma | ((x, \ell^{p_1}) :: \sigma, v_1) \in t(p_1)\}]$$
$$\cup\, [p \mapsto \bigcup_{\sigma \in t(p)} \bigcup_{((x, \ell^{p_1}) :: \sigma, v_1) \in t(p_1)} \{(\sigma, (x, \mu\ell^{p_1}.v_1) :: \sigma_2) | ((x, \mu\ell^{p_1}.v_1) :: \sigma, \sigma_2) \in t(p_2)\}]$$

The collecting semantics $p_0\Sigma_0$ computed by

$$p_0\Sigma_0 \triangleq \mathrm{lfp}(\lambda t.\mathrm{Step}(t) \cup t_{\mathrm{init}}) \quad \text{where } t_{\mathrm{init}} = [p_0 \mapsto \Sigma_0]$$

contains all derivations of the form $\sigma_0 \vdash p_0 \Downarrow v_0$ for some $\sigma_0 \in \Sigma_0$ and v_0. That is, (σ, v) is contained in $p_0\Sigma_0(p)$ if and only if $\sigma \vdash p \Downarrow v$ is contained in some derivation for the judgment $\sigma_0 \vdash p_0 \Downarrow v_0$.

5.2 Abstract semantics

Abstract shadow	$S^\#$	$\in$	$\mathrm{Shadow}^\#$
Abstract environment	$\sigma^\#$	$\in$	$\mathrm{Env}^\# \triangleq (\mathrm{Var} \xrightarrow{\mathrm{fin}} \mathcal{P}(\mathbb{P})) \times \mathcal{P}(\mathrm{Shadow}^\#)$
Abstract closure	$\langle \lambda x.p, p' \rangle$	$\in$	$\mathrm{Clos}^\# \triangleq \mathrm{Var} \times \mathbb{P} \times \mathbb{P}$
Abstract value	$v^\#$	$\in$	$\mathrm{Val}^\# \triangleq \mathrm{Env}^\# \times \mathcal{P}(\mathrm{Clos}^\#)$
Abstract semantics	$t^\#$	$\in$	$\mathbb{T}^\# \triangleq \mathbb{P} \to \mathrm{Env}^\# \times \mathrm{Val}^\#$
Abstract shadow	$S^\#$	$\to$	$\mathsf{Init}^\# \mid \mathsf{Read}^\#(p, x) \mid \mathsf{Call}^\#(p, p)$

$\boxed{\sigma \preceq (\sigma^\#, t^\#)}$

$$\textsc{Conc-Nil}\ \frac{}{\bullet \preceq \sigma^\#} \qquad \textsc{Conc-ENil}\ \frac{S \preceq (\sigma^\#, \varnothing)}{[S] \preceq \sigma^\#} \qquad \textsc{Conc-ConsLoc}\ \frac{p \in \sigma^\#.1(x) \qquad \sigma \preceq \sigma^\#}{(x, \ell^p) :: \sigma \preceq \sigma^\#}$$

$$\textsc{Conc-ConsWVal}\ \frac{p \in \sigma^\#.1(x) \qquad w \preceq t^\#(p).2 \qquad \sigma \preceq \sigma^\#}{(x, w) :: \sigma \preceq \sigma^\#}$$

$\boxed{w \preceq (v^\#, t^\#)}$

$$\textsc{Conc-Clos}\ \frac{\langle \lambda x.p, p' \rangle \in v^\#.2 \qquad \sigma \preceq t^\#(p').1}{\langle \lambda x.p, \sigma \rangle \preceq v^\#} \qquad \textsc{Conc-Rec}\ \frac{v \preceq t^\#(p).2 \qquad v \preceq v^\#}{\mu\ell^p.v \preceq v^\#}$$

$$\textsc{Conc-Init}\ \frac{\mathsf{Init}^\# \in v^\#.1.2}{\mathsf{Init} \preceq v^\#} \qquad \textsc{Conc-Read}\ \frac{\mathsf{Read}^\#(p, x) \in v^\#.1.2 \qquad [S] \preceq t^\#(p).1}{\mathsf{Read}(S, x) \preceq v^\#}$$

$$\textsc{Conc-Call}\ \frac{\mathsf{Call}^\#(p_1, p_2) \in v^\#.1.2 \qquad S \preceq t^\#(p_1).2 \qquad v \preceq t^\#(p_2).2}{\mathsf{Call}(S, v) \preceq v^\#}$$

Figure 5: The concretization relation between weak values and abstract values. $t^\#$ is omitted for brevity.

The concretization function $\gamma \in \mathbb{T}^\# \to \mathbb{T}$ is defined as:

$$\gamma(t^\#) \triangleq \lambda p.\{\sigma | \sigma \preceq (t^\#(p).1, t^\#)\} \cup \{(\sigma, v) | v \preceq (t^\#(p).2, t^\#)\}$$

where $\preceq$ is the concretization relation inductively defined in Figure 5.

Now the abstract semantic function can be given.

$$\boxed{\mathrm{Step}^\# : \mathbb{T}^\# \to \mathbb{T}^\#}$$

$$\mathrm{Step}^\#(t^\#) \triangleq \bigsqcup_{p \in \mathbb{P}} \mathrm{step}^\#(t^\#, p)$$

$$\boxed{\mathrm{step}^\# : (\mathbb{T}^\# \times \mathbb{P}) \to \mathbb{T}^\#}$$

$$\begin{aligned}
\mathrm{step}^\#(t^\#, p) &\triangleq [p \mapsto \bigsqcup_{p' \in t^\#(p).1.1(x)} (\bot, t^\#(p').2)] && \text{when } \{p : x\} \\
&\sqcup [p \mapsto (\bot, (([], \{\mathsf{Read}^\#(p, x)\}), \varnothing))] \text{if } t^\#(p).1.2 \neq \varnothing \\
\mathrm{step}^\#(t^\#, p) &\triangleq [p \mapsto (\bot, (\bot, \{\langle \lambda x.p', p \rangle\}))] && \text{when } \{p : \lambda x.p'\} \\
\mathrm{step}^\#(t^\#, p) &\triangleq [p_1 \mapsto (t^\#(p).1, \bot)] \text{when } \{p : p_1\ p_2\} \\
&\sqcup [p_2 \mapsto (t^\#(p).1, \bot)] \\
&\sqcup \bigsqcup_{\langle \lambda x.p', p'' \rangle \in t^\#(p_1).2.2} [p' \mapsto (t^\#(p'').1 \sqcup ([x \mapsto \{p_2\}], \varnothing), \bot)] \\
&\sqcup [p \mapsto \bigsqcup_{\langle \lambda x.p', _ \rangle \in t^\#(p_1).2.2} (\bot, t^\#(p').2)] \\
&\sqcup [p \mapsto (\bot, (([], \{\mathsf{Call}^\#(p_1, p_2)\}), \varnothing))] && \text{if } t^\#(p_1).2.1.2 \neq \varnothing \\
\mathrm{step}^\#(t^\#, p) &\triangleq [p_1 \mapsto (t^\#(p).1, \bot)] && \text{when } \{p : p_1 \rtimes p_2\} \\
&\sqcup [p_2 \mapsto (t^\#(p_1).2.1, \bot)] \\
&\sqcup [p \mapsto (\bot, t^\#(p_2).2)] \\
\mathrm{step}^\#(t^\#, p) &\triangleq \bot && \text{when } \{p : \varepsilon\} \\
\mathrm{step}^\#(t^\#, p) &\triangleq [p_1 \mapsto (t^\#(p).1 \sqcup ([x \mapsto \{p_1\}], \varnothing), \bot)] && \text{when } \{p : x = p_1; p_2\} \\
&\sqcup [p_2 \mapsto (t^\#(p).1 \sqcup ([x \mapsto \{p_1\}], \varnothing), \bot)] \\
&\sqcup [p \mapsto (\bot, (t^\#(p_2).2.1 \sqcup ([x \mapsto \{p_1\}], \varnothing), \varnothing))]
\end{aligned}$$

The abstract semantics $t^\#$ computed by

$$p_0{}^\#(\sigma_0^\#, t_0^\#) \triangleq \mathrm{lfp}(\lambda t^\#.\mathrm{Step}^\#(t^\#) \sqcup t_{\mathrm{init}}^\#) \quad \text{where } t_{\mathrm{init}} = t_0^\# \sqcup [p_0 \mapsto (\sigma_0^\#, \bot)]$$

is a sound abstraction of $p_0\Sigma_0$ when $\Sigma_0 \subseteq \gamma(\sigma_0^\#, t_0^\#)$.

5.3 Abstract linking

Now we define a sound linking operator that abstracts $\rtimes$. Assume we have

$$\sigma_0 \preceq (\sigma_0^\#, t_0^\#) \quad t \subseteq \gamma(t^\#)$$

we define:

$$\sigma_0 \rtimes t \triangleq \lambda p. \bigcup_{\sigma \in t(p)} (\sigma_0 \rtimes \sigma) \cup \bigcup_{(\sigma,v) \in t(p)} \{(\sigma_+, v_+) | \sigma_+ \in \sigma_0 \rtimes \sigma \text{ and } v_+ \in \sigma_0 \rtimes v\}$$

We want to define $\rtimes^\#$ so that the following holds:

$$\sigma_0 \rtimes t \subseteq \gamma((\sigma_0^\#, t_0^\#) \rtimes^\# t^\#)$$

This is equivalent to saying that the linked result $t_+^\# = (\sigma_0^\#, t_0^\#) \rtimes^\# t^\#$ satisfies:

$$\sigma_0 \preceq (\sigma_0^\#, t_0^\#) \text{ and } w \preceq (v^\#, t^\#) \Rightarrow w_+ \preceq (v_+^\#, t_+^\#)$$

for each $w_+ \in \sigma_0 \rtimes w$ and $p \in \mathbb{P}$, when $[v^\#, v_+^\#] = [(t^\#(p).1, \varnothing), (t_+^\#(p).1, \varnothing)]$ or $[t^\#(p).2, t_+^\#(p).2]$.

The condition for $t_+^\#$ can be deduced by attempting the proof of the above in advance.

We proceed by induction on the derivation for

$$w_+ \in \sigma_0 \rtimes w$$

and inversion on $w \preceq (v^\#, t^\#)$.

When:	$w = \mathsf{Init}$,	
Have:	$\mathsf{Init}^\# \in v^\#.1.2$	
Need:	$v_+^\# \sqsupseteq \sigma_0^\#$	
	$t_+^\# \sqsupseteq t_0^\#$	
When:	$w = \mathsf{Read}(S, x)$,	
Have:	$\mathsf{Read}^\#(p', x) \in v^\#.1.2$ and $[S] \preceq t^\#(p').1$	
Need:	$v_+^\# \sqsupseteq t_+^\#(p'').2$	for $p'' \in t_+^\#(p').1.1(x)$
	$v_+^\# \sqsupseteq ((\lceil\rceil, \{\mathsf{Read}^\#(p', x)\}), \varnothing)$	if $t_+^\#(p').1.2 \neq \varnothing$
When:	$w = \mathsf{Call}(S, v)$,	
Have:	$\mathsf{Call}^\#(p_1, p_2) \in v^\#.1.2$ and $S \preceq t^\#(p_1).2$ and $v \preceq t^\#(p_2).2$	
Need:	$v_+^\# \sqsupseteq t_+^\#(p').2$	for $\langle \lambda x.p', p'' \rangle \in t_+^\#(p_1).2.2$
	$v_+^\# \sqsupseteq ((\lceil\rceil, \{\mathsf{Call}^\#(p_1, p_2)\}), \varnothing)$	if $t_+^\#(p_1).2.1.2 \neq \varnothing$
	$t_+^\#(p') \sqsupseteq (t_+^\#(p'').1 \sqcup ([x \mapsto \{p_2\}], \varnothing), \varnothing)$	for $\langle \lambda x.p', p'' \rangle \in t_+^\#(p_1).2.2$
	$t_+^\# \sqsupseteq \mathrm{Step}^\#(t_+^\#)$	
When:	$w = (x, \ell^{p'}) :: \sigma$,	
Have:	$p' \in v^\#.1.1(x)$ and $\sigma \preceq v^\#$	
Need:	$v_+^\#.1.1(x) \ni p'$	
When:	$w = (x, w') :: \sigma$,	
Have:	$p' \in v^\#.1.1(x)$ and $w' \preceq t^\#(p').2$ and $\sigma \preceq v^\#$	
Need:	$v_+^\#.1.1(x) \ni p'$	
When:	$w = \langle \lambda x.p', \sigma \rangle$,	
Have:	$\langle \lambda x.p', p'' \rangle \in v^\#.2$ and $\sigma \preceq t^\#(p'').1$	
Need:	$v_+^\#.2 \ni \langle \lambda x.p', p'' \rangle$	

The above conditions can be summarized by saying $t_+^\#$ is a post-fixed point of:

$$\lambda t_+^\#.\mathrm{Step}^\#(t_+^\#) \sqcup \mathrm{Link}^\#(\sigma_0^\#, t^\#, t_+^\#) \sqcup t_0^\#$$

where $\mathrm{Link}^\#(\sigma_0^\#, t^\#, t_+^\#)$ is the least function that satisfies:

Let $\mathrm{link}^\# = \mathrm{Link}^\#\ (\sigma_0^\#, t^\#, t_+^\#)$ in		
For each $p \in \mathbb{P}$, when $v^\#, v_+^\# = (t^\#(p).1, \varnothing), (\mathrm{link}^\#(p).1, \varnothing)$		
or when $v^\#, v_+^\# = t^\#(p).2, \mathrm{link}^\#.2$		
If:	$\mathsf{Init}^\# \in v^\#.1.2$	
Then:	$v_+^\# \sqsupseteq \sigma_0^\#$	
If:	$\mathsf{Read}^\#(p', x) \in v^\#.1.2$	
Then:	$v_+^\# \sqsupseteq t_+^\#(p'').2$	for $p'' \in t_+^\#(p').1.1(x)$
	$v_+^\# \sqsupseteq ((\lceil\rceil, \{\mathsf{Read}^\#(p', x)\}), \varnothing)$	if $t_+^\#(p').1.2 \neq \varnothing$
If:	$\mathsf{Call}^\#(p_1, p_2) \in v^\#.1.2$	
Then:	$v_+^\# \sqsupseteq t_+^\#(p').2$	for $\langle \lambda x.p', p'' \rangle \in t_+^\#(p_1).2.2$
	$v_+^\# \sqsupseteq ((\lceil\rceil, \{\mathsf{Call}^\#(p_1, p_2)\}), \varnothing)$	if $t_+^\#(p_1).2.1.2 \neq \varnothing$
	$\mathrm{link}^\#(p') \sqsupseteq (t_+^\#(p'').1 \sqcup ([x \mapsto \{p_2\}], \varnothing), \varnothing)$	for $\langle \lambda x.p', p'' \rangle \in t_+^\#(p_1).2.2$
If:	$p' \in v^\#.1.1(x)$	
Then:	$v_+^\#.1.1(x) \ni p'$	
If:	$p' \in v^\#.1.1(x)$	
Then:	$v_+^\#.1.1(x) \ni p'$	
If:	$\langle \lambda x.p', p'' \rangle \in v^\#.2$	
Then:	$v_+^\#.2 \ni \langle \lambda x.p', p'' \rangle$	

Note that the left-hand side contains only $\mathrm{link}^\#$ and the right-hand side does not depend on the value of $\mathrm{link}^\#$.

Some auxiliary lemmas:

Lemma 5.1 (Substitution of values)**.**

$$w \preceq (v^\#, t^\#) \text{ and } u \preceq (t^\#(p).2, t^\#) \Rightarrow w[u/\ell^p] \preceq (v^\#, t^\#)$$

Lemma 5.2 (Sound $\mathrm{step}^\#$)**.**

$$\forall p, t, t^\# : t \subseteq \gamma(t^\#) \Rightarrow \mathrm{step}(t, p) \cup t \subseteq \gamma(\mathrm{step}^\#(t^\#, p) \sqcup t^\#)$$

Lemma 5.3 (Sound $\mathrm{Step}^\#$)**.**

$$\forall t_{\mathrm{init}}, t^\# : t_{\mathrm{init}} \subseteq \gamma(t^\#) \text{ and } \mathrm{Step}^\#(t^\#) \sqsubseteq t^\# \Rightarrow \mathrm{lfp}(\lambda t.\mathrm{Step}(t) \cup t_{\mathrm{init}}) \subseteq \gamma(t^\#)$$

6 Conclusion