Semantics for Exception Evaluation

서울대학교 전기·정보공학부 이재호, 이준협

December 30, 2022

1 Semantic domains

$$
\begin{array}{rcrcll}
\sigma & \in & \mathit{Env} & = & \mathit{Id} \to \mathit{Val} & \text{environment} \\
M & \in & \mathit{Mem} & = & \mathit{Loc} \to \mathit{Val} & \text{memory} \\
v & \in & \mathit{Val} & = & \mathit{Prim} + \mathit{Closure} + \mathit{Arr} + \mathit{Lbl} & \text{values} \\
\pi & \in & \mathit{Prim} & = & \{+, -, \texttt{raise}, \cdots\} & \text{primitive operators} \\
c & \in & \mathit{Const} & = & \mathbb{Z} + \mathbb{R} + \mathbb{B} + \cdots & \text{contants} \\
x & \in & \mathit{Id} & = & \text{identifiers in } \wp & \text{identifiers} \\
f & \in & \mathit{Closure} & = & \mathit{Expr} \times \mathit{Env} & \text{functions} \\
[\ell_1; \ldots; \ell_m] & \in & \mathit{Arr} & = & \mathit{Loc}^* & \text{array-like data (arrays, records, modules, etc.)} \\
\kappa & \in & \mathit{Ctor} & = & \kappa \text{ in } \wp & \text{constructors} \\
\langle \kappa, [\ell_1; \ldots; \ell_m] \rangle & \in & \mathit{Lbl} & = & \mathit{Ctor} \times \mathit{Arr} & \text{labeled data (variants, exns, etc.)} \\
\underline{\langle \kappa, [\ell_1; \ldots; \ell_m] \rangle} & \in & \underline{\mathit{Lbl}} & = & \mathit{Lbl} & \text{raised exceptions} \\
\ell & \in & \mathit{Loc} & & & \text{location}
\end{array}
$$

$$
\begin{array}{llcll}
\textit{expression} & e & \in & \mathit{Expr} & \\
 & e & ::= & \pi & \text{primitive operator} \\
 & & | & c & \text{constant} \\
 & & | & x & \text{id} \\
 & & | & \kappa \;\; e & \text{labeled data} \\
 & & | & \lambda \;\; (p \;\; e)^+ & \text{function} \\
 & & | & e \;\; e & \text{application} \\
 & & | & [e^*] & \text{array-like data} \\
\textit{pattern} & p & ::= & p_v & \text{value pattern} \\
 & & | & \underline{p_v} & \text{“computation” pattern} \\
\textit{value pattern} & p_v & ::= & \_ & \text{wildcard} \\
 & & | & x & \text{variable} \\
 & & | & \kappa \;\; p & \text{labeled pattern} \\
 & & | & [p^*] & \text{array-like data} \\
 & & | & c & \text{constant}
\end{array}
$$

$$\mathscr{E} : \underbrace{Expr \to Env \times Mem \to (Val + Exn) \times Mem}_{D}$$
$$\mathscr{F} : D \to D$$
$$\mathscr{E} = \text{fix}\,\mathscr{F}$$

1.1 One *Arr* to rule them all

1.1.1 Custom record

```
type myty = {a: int, mutable b: int}
let x = {a: 0, b: 0}
let y = x
y.b = 1
// x.b & y.b both 1
```

$$\begin{aligned}
\sigma(\mathtt{x}) &= \ell_x \\
\sigma(\mathtt{y}) &= \ell_y \\
s(\ell_{\mathtt{x}}) &= [\ell_{\mathtt{a}}, \ell_{\mathtt{b}}] \\
s(\ell_{\mathtt{y}}) &= [\ell_{\mathtt{a}}, \ell_{\mathtt{b}}] \\
s(\ell_{\mathtt{a}}) &= 0 \\
s(\ell_{\mathtt{b}}) &= 0 \rightsquigarrow 1
\end{aligned}$$

1.1.2 Exception reference

```
exception Exn1(int)
exception Exn2
let x = ref(Exn1 0)
let y = x
y := Exn2
```

$$\begin{aligned}
\sigma(\mathtt{x}) &= \ell_x \\
\sigma(\mathtt{y}) &= \ell_y \\
s(\ell_{\mathtt{x}}) = s(\ell_{\mathtt{y}}) &= [\ell_{\mathtt{contents}}] \\
s(\ell_{\mathtt{contents}}) &= \langle \mathtt{Exn1}, [\ell_\iota] \rangle \rightsquigarrow \langle \mathtt{Exn2}, [] \rangle \\
s(\ell_\iota) &= 0
\end{aligned}$$

1.1.3 Module

```
module type MySig = {
  let a: int
}
module MyMod = {
  let a = 0
}
module MyFun = (M: MySig) => {
  let b = M.a + 1
}
module M = MyFun(MyMod)
let {b: x} = module(M)
```

$$\begin{aligned}
\sigma(\mathtt{MyMod}) &= \ell_{\mathtt{MyMod}} \\
\sigma(\mathtt{M}) &= \ell_{\mathtt{M}} \\
\sigma(\mathtt{x}) &= \ell_{\mathtt{x}} \\
s(\ell_{\mathtt{MyMod}}) &= [\ell_{\mathtt{a}}] \\
s(\ell_{\mathtt{M}}) &= [\ell_{\mathtt{b}}] \\
s(\ell_{\mathtt{x}}) &= 1 \\
s(\ell_{\mathtt{a}}) &= 0 \\
s(\ell_{\mathtt{b}}) &= 1
\end{aligned}$$

```
var MyMod = {a: 0};
function MyFun(M) {
  var b = M.a + 1 | 0;
  return {b: b};
}
var b = 1;
var M = {b: b};
var x = b;
```

2 $\mathscr{C}$ to $\mathscr{G}$

We first define the set expressions that represent sets of values an arbitrary expression can have. Note that the set expressions are divided into two kinds, one for propagating function applications and another for propagating filtered patterns.

"Value" set expressions	v	$::=$	$\top$	*top*
		$\mid$	c	*constant*
		$\mid$	$\mathscr{X}$	*set variable*
		$\mid$	π	*primitive operator*
		$\mid$	$\lambda x.e$	*function*
		$\mid$	$\pi_{v\mid p}[(-\mid\mathscr{X})^*]$	*primitive application*
		$\mid$	$\mathsf{app}_{v\mid p}(\mathscr{X}, [(-\mid\mathscr{X})^*])$	*function application*
		$\mid$	$\langle(-\mid\kappa), [\ell^*]\rangle$	*variants, records, tuples, arrays*
		$\mid$	$\mathsf{fld}(\mathscr{X}, (-\mid\kappa), i)$	*field*
		$\mid$	$\mathscr{X} - p$	*pattern filtering*
"Pattern" set expressions	p	$::=$	$\hat{\top}$	*top*
		$\mid$	$\hat{c}$	*constant*
		$\mid$	$\langle(-\mid\kappa), [p^*]\rangle$	*variants, records, tuples, arrays*
		$\mid$	$\hat{\ell}$	*location*
		$\mid$	$\hat{\ell}\mid_p$	*location constrained by p*

Next the structure of possible set constraints are illustrated. The constraints collect what each set variable will contain, what each update will do, and what each location will contain.

Constraints $\mathscr{C}$	$\mathscr{X}$	$\supseteq$	v	*set variables*
	$\mathsf{fld}(\mathscr{X}, -, i)$	$\supseteq$	v	*updates*
	$!\ell$	$\supseteq$	v	*variant, record, tuple, array elements*
Grammar $\mathscr{G}$	$\hat{\mathscr{X}}$	$\supseteq$	p	*set variables*
	$!\hat{\ell}$	$\supseteq$	p	*variant, record, tuple, array elements*

Note that $\langle -, [\ell^*]\rangle$ is used to represent the *Arr* datatype in section 1.

We need to find the least fixpoint $\mathsf{lfp}(\lambda(C,G).\mathscr{F}(\mathscr{C} \cup C, G)) =: (\mathscr{C}', \mathscr{G})$. $\mathscr{F}(C,G)$ takes a set of constraints C and a grammar G and performs "one step of resolution" to return a partially-resolved (C', G'). $\mathscr{C}$ is the initial set of constraints obtained from the program.

What we want is a good definition of $\mathscr{F}$ so that $(C,G) \sqsubseteq \mathscr{F}(C,G)$ and $\mathscr{F}^\infty(C,G)$ converges surely while safely approximating all possible values. $\perp_C$ and $\perp_G$ is defined as the empty constraint/grammar. Note that a production in G specifies some pattern that $\hat{\mathscr{X}}$ and $!\hat{\ell}$ might match.

2.1 Definition of $\mathscr{F}$

$\mathscr{F}(C, G)$: Look at the "productions" in C, determine what can be added to C and G.

Preliminaries:

$$\mathsf{len}(l) := \begin{cases} 0 & (l = []) \\ \mathsf{len}(\mathsf{tl}(l)) & (\mathsf{hd}(l) = -) \\ \mathsf{len}(\mathsf{tl}(l)) + 1 & (\mathsf{hd}(l) \neq -) \end{cases}$$

$$\mathsf{merge}(l, l') := \begin{cases} l' & (l = []) \\ l & (l' = []) \\ \mathsf{hd}(l') :: \mathsf{merge}(\mathsf{tl}(l), \mathsf{tl}(l')) & (\mathsf{hd}(l) = -) \\ \mathsf{hd}(l) :: \mathsf{merge}(\mathsf{tl}(l), l') & (\mathsf{hd}(l) \neq -) \end{cases}$$

That is, $\mathsf{merge}(l, l')$ is performed by plugging in elements of l' one by one into the $-$ places of l(including $- \in l'$), then concatenating the rest of l' to the tail side of l if there is no more free space.

Now, let's define $\mathscr{F}$:

1. For productions $\mathscr{X}|!\ell \supseteq \top \mid c \mid | \langle(-|\kappa), [\ell^*]\rangle$, add the same productions "with a hat" to G if they are not already in G.
2. For production $\mathscr{X}|!\ell \supseteq \mathscr{X}_1$,
   (a) If $\hat{\mathscr{X}}_1 \supseteq \star$ is in G, add $\hat{\mathscr{X}}|!\hat{\ell} \supseteq \star$ to G.
   (b) If $\mathscr{X}_1 \supseteq \pi \mid \lambda x.e \mid \mathsf{app}_v(\mathscr{X}_2, - :: \mathtt{tl}) \mid \pi_v\ \mathtt{l}$ is in C, add those to $\mathscr{X}|!\ell$ in C.
3. For production $\mathscr{X}|!\ell \supseteq \pi_v\ \mathtt{l}$, when $\mathsf{len}(\mathtt{l}) = \mathsf{arity}(\pi)$ and π is not `raise`, add $\hat{\mathscr{X}}|!\hat{\ell} \supseteq \top$ to G (constant propagation may be added). Note that `ignore`, `identity`, `reverse` must also be put into consideration, but this is trivial.
4. For production $\mathscr{X}|!\ell \supseteq \pi_p\ \mathtt{l}$, when $\mathsf{len}(\mathtt{l}) = \mathsf{arity}(\pi)$ and π is `raise`, add $\hat{\mathscr{X}}|!\hat{\ell} \supseteq \mathsf{hd}(\mathtt{l})$ to G.
5. For production $\mathscr{X}|!\ell \supseteq \mathsf{app}_v(\mathscr{X}_1, [])$, this only happens on a `Lazy.force`, so if $\mathscr{X}_1 \supseteq \lambda.e$ is in C, then add $\hat{\mathscr{X}}|!\hat{\ell} \supseteq \mathscr{X}(e)$ to G.
6. For production $\mathscr{X}|!\ell \supseteq \mathsf{app}_p(\mathscr{X}_1, [])$, this only happens on a `Lazy.force`, so if $\mathscr{X}_1 \supseteq \lambda.e$ is in C, then add $\hat{\mathscr{X}} \supseteq \mathscr{X}|!\hat{\ell}(\underline{e})$ to G.
7. For production $\mathscr{X}|!\ell \supseteq \mathsf{app}_v(\mathscr{X}_1, \mathscr{X}_2 :: \mathtt{tl})$,
   (a) If $\mathscr{X}_1 \supseteq \pi$ is in C, add $\mathscr{X}|!\ell \supseteq \pi_v\ \mathscr{X}_2 :: \mathtt{tl}$ to C.
   (b) If $\mathscr{X}_1 \supseteq \pi_v\ \mathtt{l}$ is in C, add $\mathscr{X}|!\ell \supseteq \pi_v\ \mathsf{merge}(\mathtt{l}, \mathscr{X}_2 :: \mathtt{tl})$ to C.
   (c) If $\mathscr{X}_1 \supseteq \lambda x.e$ is in C, $\mathtt{tl} \neq []$, add $\mathscr{X}|!\ell \supseteq \mathsf{app}_v(\mathscr{X}(e), \mathtt{tl})$, $\mathscr{X}(E_x) \supseteq \mathscr{X}_1$ to C.
   (d) If $\mathscr{X}_1 \supseteq \lambda x.e$ is in C, $\mathtt{tl} = []$, add $\mathscr{X}|!\ell \supseteq \mathscr{X}(e)$, $\mathscr{X}(E_x) \supseteq \mathscr{X}_1$ to C.
   (e) If $\mathscr{X}_1 \supseteq \mathsf{app}_v(\mathscr{X}_3, - :: \mathtt{tl}')$ is in C, add $\mathscr{X}|!\ell \supseteq \mathsf{app}_v(\mathscr{X}_3, \mathscr{X}_2 :: \mathsf{merge}(\mathtt{tl}, \mathtt{tl}'))$ to C.
8. For production $\mathscr{X}|!\ell \supseteq \mathsf{app}_p(\mathscr{X}_1, \mathscr{X}_2 :: \mathtt{tl})$,
   (a) If $\mathscr{X}_1 \supseteq \pi$ is in C, add $\mathscr{X}|!\ell \supseteq \pi_p\ \mathscr{X}_2 :: \mathtt{tl}$ to C.

(b) If $\mathcal{X}_1 \supseteq \pi_v\ \mathtt{l}$ is in C, add $\mathcal{X}|!\ell \supseteq \pi_p\ \mathsf{merge}(\mathtt{l}, \mathcal{X}_2 :: \mathtt{tl})$ to C.

(c) If $\mathcal{X}_1 \supseteq \lambda x.e$ is in C, add $\mathcal{X}|!\ell \supseteq \mathcal{X}(\underline{e})$, $\mathcal{X}(E_x) \supseteq \mathcal{X}_1$ to C. Additionally, if $\mathtt{tl} \neq []$ then also add $\mathsf{app}_p(\mathcal{X}(e), \mathtt{tl})$.

(d) If $\mathcal{X}_1 \supseteq \mathsf{app}_v(\mathcal{X}_3, - :: \mathtt{tl}')$ is in C, add $\mathcal{X}|!\ell \supseteq \mathsf{app}_p(\mathcal{X}_3, \mathcal{X}_2 :: \mathsf{merge}(\mathtt{tl}, \mathtt{tl}'))$ to C.

9. For production $\mathcal{X}|!\ell \supseteq \mathsf{fld}(\mathcal{X}_1, -, i)$,

   (a) If $\hat{\mathcal{X}}_1 \supseteq \top$ is in G, add $\hat{\mathcal{X}}|!\hat{\ell} \supseteq \top$ to G.

   (b) If $\hat{\mathcal{X}}_1 \supseteq \langle(-|\kappa), ...\hat{\ell}_i...\rangle$ is in G and if $!\hat{\ell}_i \supseteq$ something is in G, add $\hat{\mathcal{X}}|!\hat{\ell} \supseteq$ something to G.

   (c) If $\hat{\mathcal{X}}_1 \supseteq \langle(-|\kappa), ...p_i...\rangle$ is in G, add $\hat{\mathcal{X}}|!\hat{\ell} \supseteq p_i$ to G.

10. For production $\mathcal{X}|!\ell \supseteq \mathsf{fld}(\mathcal{X}_1, \kappa, i)$,

    (a) If $\hat{\mathcal{X}}_1 \supseteq \top$ is in G, add $\hat{\mathcal{X}}|!\hat{\ell} \supseteq \top$ to G.

    (b) If $\hat{\mathcal{X}}_1 \supseteq \langle\kappa, ...\hat{\ell}_i...\rangle$ is in G and if $!\hat{\ell}_i \supseteq$ something is in G, add $\hat{\mathcal{X}}|!\hat{\ell} \supseteq$ something to G. The constructor κ must be matched.

    (c) If $\hat{\mathcal{X}}_1 \supseteq \langle\kappa, ...p_i...\rangle$ is in G, add $\hat{\mathcal{X}}|!\hat{\ell} \supseteq p_i$ to G.

11. For production $\mathsf{fld}(\mathcal{X}, -, i) \supseteq$ something,

    (a) If $\hat{\mathcal{X}}_1 \supseteq \langle(-|\kappa), ...\hat{\ell}_i...\rangle$ is in G, add $!\ell_i \supseteq$ something to C.

12. For production $\mathcal{X}|!\ell \supseteq \mathcal{X}_1 - p$, first define

$$\mathsf{filter}(x, p) := \begin{cases} \varnothing & (p = \top) \\ \varnothing & (x = p) \\ \{x\} & (x \neq p, p = c) \\ \{x\} & (x = \langle\kappa, -\rangle, p = \langle\kappa', -\rangle) \\ \mathsf{filter}(\langle -, [\underbrace{\top; ...; \top}_{n \text{ times}}]\rangle, p) & (x = \top, p = \langle -, [p_1; ...; p_n]\rangle) \\ \bigcup_{\hat{x} \supseteq y \in G} \mathsf{filter}(y, p) & ((x, \hat{x}) = (\hat{\mathcal{X}}, \hat{\mathcal{X}}) \text{ or } (\hat{\ell}, !\hat{\ell})) \\ \bigcup_{i=0}^{n-1} \left\langle \kappa, \left(\prod_{j=1}^{i} p_j\right) \mathsf{filter}(x_{i+1}, p_{i+1}) \left(\prod_{j=i+2}^{n} x_j\right) \right\rangle & \underbrace{(x = \langle\kappa, [x_1; ...; x_n]\rangle, p = \langle\kappa, [p_1; ...; p_n]\rangle)}_{\kappa \text{ may be } -} \end{cases}$$

    when $x \in \{p, \hat{\ell}, \hat{\mathcal{X}}\}$

    Note that $\top$ happens on a π_v, and the type of p must match with the type of x, so we can reconstruct the type of $\top$ from p (when we use externals that return records). Also, in the list concatenation part of the last case, if the result of filter is $\varnothing$, the whole thing is $\varnothing$.

    Now, add $\hat{\mathcal{X}}|!\hat{\ell} \supseteq \alpha$ for all $\alpha \in \mathsf{filter}(\hat{\mathcal{X}}_1, p)$ to G.

2.2 `Array.make` (resolution of π: maybe later)

For primitives π, a new constructor is generated, e.g., $\mathsf{app}_v(\pi, [])$ to $\mathsf{ctor}(-, [\ell_{\text{new}}])$.

2.3 Brainstorming

$$\begin{aligned}
\langle \kappa, \ell_1 \ell_2 ... \ell_n \rangle - \langle \kappa, p_1 p_2 ... p_n \rangle &= \langle \kappa, (\ell_1 - p_1) \ell_2 ... \ell_n \rangle \\
&\quad + \langle \kappa, p_1 (\ell_2 ... \ell_n - p_2 ... p_n) \rangle \\
&= ... \\
&= \sum_{i=0}^{n-1} \left(\prod_{j=1}^{i} p_j \right) (\ell_{i+1} - p_{i+1}) \left(\prod_{j=i+2}^{n} \ell_j \right) \\
\ell - p &=! \hat{\ell} - p
\end{aligned}$$

Here, the product notation stands for list concatenation, when p_i and ℓ_i stands for a single-element list. $\ell - p$ stands for the contents of the abstract location ℓ that is not matched by the pattern p : how to express this neatly?