Simplified Logical Relation for FIP and Perceus

ANTON LORENZEN, University of Edinburgh, UK

1 PURE SEMANTICS

We define the syntax in Figure 1. To illustrate the key ideas, we use a simplified language where only lambdas are available.

Expressions:

$$
\begin{array}{llll}
e & ::= & v & \text{(values)} \\
 & | & e\ v & \text{(application)} \\
 & | & \mathsf{let}\ x\ =\ e\ \mathsf{in}\ e & \text{(let binding)} \\
 & | & \mathsf{inc}\ v;\ e & \text{(rc increment)} \\
 & | & \mathsf{dec}\ v;\ e & \text{(rc decrement)}
\end{array}
\qquad
\begin{array}{llll}
v & ::= & x, y & \text{(variables)} \\
 & | & \lambda x.\ e & \text{(lambda)}
\end{array}
$$

Fig. 1. Syntax of the λ^{FIP} calculus.

The pure big-step semantics of the FIP calculus is given in Figure 2:

$$
\frac{e_1 \Downarrow v \qquad e_2[x{:=}v] \Downarrow w}{\mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2 \Downarrow w}\ \textsc{let}
\qquad
\frac{e \Downarrow (\lambda x.\ e') \qquad e'[x{:=}v] \Downarrow w}{e\ v \Downarrow w}\ \textsc{app}
$$

$$
\frac{e \Downarrow w}{\mathsf{inc}\ v;\ e \Downarrow w}\ \textsc{inc}
\qquad
\frac{e \Downarrow w}{\mathsf{dec}\ v;\ e \Downarrow w}\ \textsc{dec}
$$

Fig. 2. Functional big-step semantics.

2 TYPING RULES

We present the rules of the simplified Perceus type system in Figure 3. We write $\Gamma \vdash e$ to say that the expression e consumes exactly the variables in Γ. All rules of the calculus are substructural, where we allow exchange but disallow contraction and weakening. However, contraction can be achieved by inserting a reference count increment and weakening using a reference count decrement. We now keep track of the free variables of lambdas in the syntax, moving from $\lambda x.\ e$ to $\lambda^{\overline{z}} x.\ e$, where $\overline{z}$ are the free variables of $\lambda x.\ e$.

3 HEAP SEMANTICS

We can prove the reference counting scheme sound using the heap semantics in Figure 4. It differs from the pure semantics in the use of a heap H which stores all bindings with their reference counts. Furthermore, the final output of this semantics is a variable x, which, when read from the heap $[\mathsf{H}]x$ gives the final result v of the program.

We write $[\mathsf{H}]x$ for the value v obtained by recursively reading the variable x from the heap H. Our soundness result states that if a program evaluates to a value v in the pure semantics, then it also evaluates to the same value $[\mathsf{H}]x$ in the heap semantics:

Corollary 1.
If $e \Downarrow v$ and $\varnothing \vdash e$, then $\varnothing \mid e \longmapsto_{\mathsf{h}}^{*} \mathsf{H} \mid x$ and $[\mathsf{H}]x\ =\ v$.

$\Gamma \quad ::= \quad \varnothing \mid \Gamma, x \quad$ (owned environment)

$$\frac{}{x \vdash x}\ \textsc{var} \qquad \frac{\Gamma_1 \vdash e_1 \quad \Gamma_2, x \vdash e_2 \quad x \notin \Gamma_2}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2}\ \textsc{let}$$

$$\frac{\overline{z}, x \vdash e \quad \overline{z} = \mathsf{fv}(\lambda x.\ e)}{\overline{z} \vdash \lambda^{\overline{z}}\ x.\ e}\ \textsc{lam} \qquad \frac{\Gamma_1 \vdash e \quad \Gamma_2 \vdash v}{\Gamma_1, \Gamma_2 \vdash e\ v}\ \textsc{app}$$

$$\frac{\Gamma, x, x \vdash e}{\Gamma, x \vdash \mathsf{inc}\ x;\ e}\ \textsc{inc} \qquad \frac{\Gamma \vdash e}{\Gamma, x \vdash \mathsf{dec}\ x;\ e}\ \textsc{dec}$$

Fig. 3. Simplified λ^{FIP} calculus

$\mathsf{H} ::= \varnothing \mid \mathsf{H}, x \mapsto^n \lambda^{\overline{z}} x'.\ e$
$\mathsf{E} ::= \square \mid \overline{x}\ \mathsf{V} \mid \mathsf{E}\ v \mid \mathsf{let}\ \overline{x} = \mathsf{E}\ \mathsf{in}\ e$

$$\frac{\mathsf{H} \mid e \longrightarrow_{\mathsf{h}} \mathsf{H}' \mid e'}{\mathsf{H} \mid \mathsf{E}[e] \longmapsto_{\mathsf{h}} \mathsf{H}' \mid \mathsf{E}[e']}\ \textsc{eval}$$

$$\begin{array}{llcll}
(lam_h) & \mathsf{H} \mid \lambda^{\overline{z}}\ x'.\ e & \longrightarrow_{\mathsf{h}} & \mathsf{H}, x \mapsto^1 \lambda^{\overline{z}} x'.\ e \mid x & (\text{fresh } x) \\
(beta_h) & \mathsf{H} \mid (f)\ y & \longrightarrow_{\mathsf{h}} & \mathsf{H} \mid \mathsf{inc}\ \overline{z};\ \mathsf{dec}\ f;\ e[x{:=}y] & (f \mapsto^n \lambda^{\overline{z}} x.\ e \in \mathsf{H}) \\
(let_h) & \mathsf{H} \mid \mathsf{let}\ x = z\ \mathsf{in}\ e & \longrightarrow_{\mathsf{h}} & \mathsf{H} \mid e[x{:=}z] &
\end{array}$$

$$\begin{array}{lllcl}
(inc_h) & \mathsf{H}, x \mapsto^n v & \mid \mathsf{inc}\ x;\ e & \longrightarrow_{\mathsf{h}} & \mathsf{H}, x \mapsto^{n+1} v \mid e \\
(dec_h) & \mathsf{H}, x \mapsto^{n+1} v & \mid \mathsf{dec}\ x;\ e & \longrightarrow_{\mathsf{h}} & \mathsf{H}, x \mapsto^n v \mid e \\
(dlam_h) & \mathsf{H}, x \mapsto^1 \lambda^{\overline{z}} x'.e' & \mid \mathsf{dec}\ x;\ e & \longrightarrow_{\mathsf{h}} & \mathsf{H} \mid \mathsf{dec}\ \overline{z};\ e
\end{array}$$

Fig. 4. Heap semantics of λ^{FIP}.

This result, which follows directly from our soundness theorem below, does not guarantee that well-typed programs never get stuck. Instead, it shows the correctness of the reference counted program *under the assumption that the pure semantics does not get stuck* and is thus fully orthogonal to any type system guaranteeing that the pure semantics can never get stuck.

4 LOGICAL RELATION

To show the soundness result, we first define the "roots" of a heap, which are the variables that do not have the correct reference counts internally. For example, if a variable has referenece count 3 but is referred only once in the heap, then we have two roots pointing to that variable. We collect roots in a function from the heap variables to $\mathbb{Z}$, where we write I_x for the indicator function of x which returns 1 for the argument x and 0 else. As usual, we use pointwise addition and multiplication on the functions:
$\mathsf{roots}(\varnothing) = 0$
$\mathsf{roots}(\mathsf{H}, x \mapsto^n v) = \mathsf{roots}(\mathsf{H}) + n * I_x - I_{z_1} - \ldots - I_{z_n}$ where $\overline{z} = \mathsf{fv}(v)$
The function $\mathsf{roots}(\mathsf{H})$ returns 0 for all variables x that have a reference count which is exactly equal to the number of times this variable is referred to in the heap. Notice also that the roots of a heap can be negative: for example, we could model the memory underlying a magic wand $x -\!\!* \ y$ as a heap with roots $x \mapsto -1,\ y \mapsto 1$. We call a heap H *linear* if $\mathsf{roots}(\mathsf{H}) \geq 0$. In that case, we can also use $\mathsf{roots}(\mathsf{H})$ as multiset (where each variable occurs as often as indicated by the function).

We call two heaps $\mathsf{H}_1, \mathsf{H}_2$ *compatible* if they map equal names $x \mapsto^n v \in \mathsf{H}_1,\ x \mapsto^m w \in \mathsf{H}_2$, to equal values $v = w$. We can join two compatible heaps using the join operator $\otimes$. This operator adds the reference counts at each variable but it carefully removes the reference counts of the children to ensure that no internal references are counted twice:

$$
\begin{array}{lcll}
\varnothing \otimes \mathsf{H}_2 & = & \mathsf{H}_2 & \\
\mathsf{H}_1, x \mapsto^n v \otimes \mathsf{H}_2 & = & \mathsf{H}_1 \otimes \mathsf{H}_2, x \mapsto^n v & \text{iff } x \notin \mathsf{dom}(\mathsf{H}_2) \\
\mathsf{H}_1, x \mapsto^n v \otimes \mathsf{H}_2, x \mapsto^m v, \overline{z \mapsto^{k+1} w} & = & \mathsf{H}_1 \otimes \mathsf{H}_2, x \mapsto^{n+m} v, \overline{z \mapsto^k w} & \text{iff } \overline{z} = \mathsf{fv}(v)
\end{array}
$$

Lemma 1. (*Heap join is associative and commutative*)
For all compatible heaps $\mathsf{H}_1, \mathsf{H}_2, \mathsf{H}_3$: $\mathsf{H}_1 \otimes (\mathsf{H}_2 \otimes \mathsf{H}_3) = (\mathsf{H}_1 \otimes \mathsf{H}_2) \otimes \mathsf{H}_3$ and $\mathsf{H}_1 \otimes \mathsf{H}_2 = \mathsf{H}_2 \otimes \mathsf{H}_1$.

Lemma 2. (*The roots of heaps are added by the join operator*)
For any two compatible heaps $\mathsf{H}_1, \mathsf{H}_2$: $\mathsf{roots}(\mathsf{H}_1 \otimes \mathsf{H}_2) = \mathsf{roots}(\mathsf{H}_1) + \mathsf{roots}(\mathsf{H}_2)$.

This is a useful property, since it justifies the use of the join operator as a form of separating conjunction for reference counted heaps. We can add a root by incrementing its reference count and remove a root by decrementing its reference count. Notice that the decrement might have to recursively decrement the reference counts of the children – but this complexity is encapsulated in the definition of a root:

Lemma 3. (*Incrementing adds a root*)
If $\mathsf{H} \mid \mathsf{inc}\ x;\ e \longmapsto \mathsf{H}' \mid e$, then $\mathsf{roots}(\mathsf{H}') = \mathsf{roots}(\mathsf{H}) + I_x$.

Lemma 4. (*Decrementing removes a root*)
If $\mathsf{H} \mid \mathsf{dec}\ x;\ e \longmapsto^{*} \mathsf{H}' \mid e$, then $\mathsf{roots}(\mathsf{H}') = \mathsf{roots}(\mathsf{H}) - I_x$.

Now we are ready to define a logical relation for the heap semantics. First, we define a denotation for values:

Definition 1. (*Value denotation*)
For any closed value v (tuple of closed values $\overline{v}$), we define the set $[\![v]\!]$ (set $[\![\overline{v}]\!]$) as:

$$[\![(v_1, \ldots, v_n)]\!] = \{\, (\mathsf{H}, (x_1, \ldots, x_n)) \mid \mathsf{H} = \mathsf{H}_1 \otimes \ldots \otimes \mathsf{H}_n \text{ and } (\mathsf{H}_i, x_i) \in [\![v_i]\!] \text{ for } i = 1, \ldots, n \,\}$$

$$
\begin{array}{l}
[\![\lambda x'.\, e]\!] = \{\, (\mathsf{H}, x) \mid \mathsf{H} = \mathsf{H}_1 \otimes (x \mapsto^1 \lambda^{\overline{z}}\, x'.\, e') \\
\quad \text{and for some values } \overline{v},\ e = e'[\overline{z}{:=}\overline{v}] \text{ and } (\mathsf{H}_1, \overline{z}) \in [\![\overline{v}]\!] \\
\quad \text{and for all } (\mathsf{H}_2, y) \in [\![w]\!] \text{ with } \mathsf{H}_1 \text{ and } \mathsf{H}_2 \text{ compatible and } e[x'{:=}w] \Downarrow w', \\
\quad \text{we have that } \mathsf{H}_1 \otimes \mathsf{H}_2 \mid e'[x'{:=}y] \longmapsto^{*}_{\mathsf{h}} \mathsf{H}_3 \mid y' \text{ and } (\mathsf{H}_3, y') \in [\![w']\!] \\
\quad \text{and } \mathsf{H}_3 \text{ is compatible with } \mathsf{H}_1, \mathsf{H}_2 \,\}
\end{array}
$$

Notice that the definition of the value denotation implies that if $(\mathsf{H}, x) \in [\![v]\!]$, then recursively reading x from the heap H will yield the value v.

Definition 2. (*Context substitution*)
We write $(\mathsf{H}, \sigma) : \Gamma$ if σ is a substitution mapping the variables $\Gamma = \overline{x}$ to values $\overline{v}$ and $(\mathsf{H}, \overline{x}) \in [\![\overline{v}]\!]$.

Building on this foundation, we can now define the logical relation for the heap semantics:

Definition 3. (*Logical relation*)
We write $\Gamma \vDash e$ if for all $(\mathsf{H}_1, \sigma) : \Gamma$ we have that $\sigma(e) \Downarrow v$ implies $\mathsf{H}_1 \mid e \longmapsto^{*}_{\mathsf{h}} \mathsf{H}_2 \mid x$ and $(\mathsf{H}_2, x) \in [\![v]\!]$ and H_2 is compatible with H_1.

We can now prove the soundness of the heap semantics:

Theorem 1. (*The heap semantics is sound for well-formed Perceus programs*)
If $\Gamma \vdash e$, then $\Gamma \vDash e$.

This soundness theorem directly implies our corollary above.