Modular Static Analysis: Orthogonal Design, Correct by Construction

$$
\begin{aligned}
x &\in \mathit{Var} \\
i &\in \mathbb{Z} \\
op \in \mathit{Op} &::= + \mid - \\
a \in \mathit{Atom} &::= x \mid i \mid \boldsymbol{\lambda}(x) \rightarrow e \\
e \in \mathit{Exp} &::= a \mid (e_1 \; op \; e_2) \mid (e_1 \;\; e_2) \mid \textbf{if0}(e_1)\{e_2\}\{e_3\}
\end{aligned}
$$

Figure 1: λ-IF syntax.

1. Modular Analysis by Example

To demonstrate our framework we will grow a concrete semantics into an executable analysis in small steps. Each step will expose some property of the analysis to be tuned independent of other properties. We use an applied lambda calculus as an example language to demonstrate our approach.

1.1 Concrete Semantics

Our example language is **λ-IF**, an applied lambda calculus with integers and conditional statements. The syntax for **λ-IF** is given in figure 1.

We will use a standard small-step semantics for **λ-IF**, which is shown in figure 2.

The analyses we will consider will be stated as the least fixed point of some abstract collecting semantics. A collecting semantics for **λ-IF** is defined as:

$$\mu(\varsigma).\varsigma_0 \sqcup \varsigma \sqcup \{\varsigma' \mid \varsigma \rightsquigarrow \varsigma'\}$$

$$
\begin{aligned}
clo \in \mathit{Clo} &:= \langle \boldsymbol{\lambda}(x) \rightarrow e, \rho \rangle \\
v \in \mathit{Val} &::= i \mid clo \\
\rho \in \mathit{Env} &:= \mathit{Var} \rightharpoonup \mathit{Val} \\
fr \in \mathit{Frame} &::= [\square \; op \; e] \mid [v \; op \; \square] \mid [\square \;\; e] \mid [v \;\; \square] \mid [\textbf{if0}(\square)\{e_1\}\{e_2\}] \\
\overline{fr} \in \mathit{Kon} &:= \mathit{Frame}^* \\
\varsigma \in \Sigma &:= \mathit{Exp} \times \mathit{Env} \times \mathit{Kon} \\
\rightsquigarrow &\in \mathcal{P}(\Sigma \times \Sigma)
\end{aligned}
$$

$$
\begin{aligned}
\langle a, \rho, [\square \; op \; e] :: \overline{fr} \rangle &\rightsquigarrow \langle e, \rho, [\mathcal{A}_\rho[\![a]\!] \; op \; \square] :: \overline{fr} \rangle \\
\langle a, \rho, [v \; op \; \square] :: \overline{fr} \rangle &\rightsquigarrow \langle \delta(op, v, \mathcal{A}_\rho[\![a]\!]), \rho, \overline{fr} \rangle \\
\langle a, \rho, [\square \;\; e] :: \overline{fr} \rangle &\rightsquigarrow \langle e, \rho, [\mathcal{A}_\rho[\![a]\!] \;\; \square] :: \overline{fr} \rangle \\
\langle a, \rho, [\langle \boldsymbol{\lambda}(x) \rightarrow e, \rho' \rangle \;\; \square] :: \overline{fr} \rangle &\rightsquigarrow \langle e, \rho'[x \mapsto \mathcal{A}_\rho[\![a]\!]], \overline{fr} \rangle \\
\langle a, \rho, [\textbf{if0}(\square)\{e_1\}\{e_2\}] :: \overline{fr} \rangle &\rightsquigarrow \begin{cases} \langle e_1, \rho, \overline{fr} \rangle & \text{if } \mathcal{A}_\rho[\![a]\!] = 0 \\ \langle e_2, \rho, \overline{fr} \rangle & \text{otherwise} \end{cases} \\
\langle e_1 \; op \; e_2, \rho, \overline{fr} \rangle &\rightsquigarrow \langle e_1, \rho, [\square \; op \; e_2] :: \overline{fr} \rangle \\
\langle e_1 \;\; e_2, \rho, \overline{fr} \rangle &\rightsquigarrow \langle e_1, \rho, [\square \;\; e_2] :: \overline{fr} \rangle \\
\langle \textbf{if0}(e_1)\{e_2\}\{e_3\}, \rho, \overline{fr} \rangle &\rightsquigarrow \langle e_1, \rho, [\textbf{if0}(\square)\{e_2\}\{e_3\}] :: \overline{fr} \rangle
\end{aligned}
$$

Figure 2: λ-IF semantics.

where $\varsigma_0 := \langle e_0, \bot, \bullet \rangle$ is the injection of the initial program e_0 into the state space Σ.

2. From Semantics to Interpreter

On our way to an executable analysis for **λ-IF**, we first evolve the small-step relation into an executable small-step interpreter. We write this interpreter in monadic style for two reasons. First, it is much easier to add new states and rules to a monadic interpreter. Second, we will exploit the monadic abstraction to expose flow and path sensitivities in a later interpreter.

The monad for the interpreter will have two types effects: state and partiality. Two operations carry state effects: get and put; and one operation carries a partiality effect: fail.

We construct a monad $\mathcal{M}$ which combines state effects for Env and Kon with partiality effects. The type $\mathcal{M}$ is just a simple type which supports the monad,

$$
\begin{aligned}
&\mathcal{M}(a) := Env \times Kon \to (a \times Env \times Kon)_{\bot} \\
&\text{atom} : Atom \to \mathcal{M}(Exp) \\
&\text{atom}(a) := \underline{\text{do}} \\
&\quad v \leftarrow \mathcal{A}[\![a]\!] \\
&\quad fr \leftarrow \text{pop-Kon} \\
&\quad \underline{\text{case}}\ fr\ \underline{\text{of}} \\
&\qquad [\square\ op\ e] \to \underline{\text{do}} \\
&\qquad\quad \text{push-Kon}([v\ op\ \square]) \\
&\qquad\quad \underline{\text{return}}(e) \\
&\qquad [v'\ op\ \square] \to \underline{\text{return}}(\delta(op, v', v)) \\
&\qquad [\square\ \ e] \to \underline{\text{do}} \\
&\qquad\quad \text{push-Kon}([v\ \ \square]) \\
&\qquad\quad \underline{\text{return}}(e) \\
&\qquad [v'\ \ \square] \to \underline{\text{do}} \\
&\qquad\quad \langle \boldsymbol{\lambda}(x) \to e, \rho' \rangle \leftarrow \text{to-clo}(v') \\
&\qquad\quad \text{put-Env}(\rho'[x \mapsto v]) \\
&\qquad\quad \underline{\text{return}}(e) \\
&\qquad [\textbf{if0}(\square)\{e_1\}\{e_2\}] \to \underline{\text{if}}\ v \stackrel{?}{=} 0\ \underline{\text{then}}\ \underline{\text{return}}(e_1)\ \underline{\text{else}}\ \underline{\text{return}}(e_2) \\
&\text{step} : Exp \to \mathcal{M}(Exp) \\
&\text{step}(a) := \text{atom}(a) \\
&\text{step}(e_1\ op\ e_2) := \underline{\text{do}} \\
&\quad \text{push-Kon}([\square\ op\ e_2]) \\
&\quad \underline{\text{return}}(e_1) \\
&\text{step}(e_1\ \ e_2) := \underline{\text{do}} \\
&\quad \text{push-Kon}([\square\ \ e_2]) \\
&\quad \underline{\text{return}}(e_1) \\
&\text{step}(\textbf{if0}(e_1)\{e_2\}\{e_3\}) := \underline{\text{do}} \\
&\quad \text{push-Kon}([\textbf{if0}(\square)\{e_2\}\{e_3\}]) \\
&\quad \underline{\text{return}}(e_1)
\end{aligned}
$$

Figure 3: λ-IF monadic interpreter.

state, and partiality operations. The interpreter for **λ-IF** which uses $\mathcal{M}$ and monadic effects is given in figure 3.

In the small-step abstract interpretation setting, an analysis is defined as the least fixed point of a collecting semantics over the step relation.

3. Exposing the Abstract Domain

We will now expose the choice of abstract domain for the interpreter. However, giving a finite abstract domain for integers alone will not yet give a computable analysis. There will still be an infinite number of possible abstract state spaces due to the recursion between closures and environments. We will address this issue in a future modification to the interpreter.

To expose the choice of abstract domain we introduce a type $AVal$ behind an abstract interface.

$$
\begin{aligned}
AVal &\in Set \\
\text{int-I} &: \mathbb{Z} \to AVal \\
\text{if0-E} &: AVal \to \mathcal{P}(\mathbb{B}) \\
\text{clo-I} &: Clo \to AVal \\
\text{clo-E} &: AVal \to \mathcal{P}(Clo) \\
\delta &: Op \to AVal \to AVal \to AVal
\end{aligned}
$$

To establish the correctness of our interpreter step, we must prove that it is monotonic in $AVal$, $intI$, $if0E$, $cloI$ and $cloE$. This can be done independent of a specific $AVal$. Then we need only establish $Val \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} AVal$ for some $AVal$ in order to justify the correctness of step instantiated with $AVal$.

Exposing the abstract domain forces us to reconsider our monadic effects. For example, a branch on a possibly unknown integer value must return multiple possible machine states. Therefore we must trade our partiality monad for a nonteterminism monad. The monadic abstraction supports nondeterminism nicely in this setting. The helper function branch-on-set will return multiple times for each value in the set. This allows the interpreter to be written in a style that need only consider one state at a time. The monadic interpreter whith the abstract domain exposd is given in figure 4. (The implementation of step is unchanged.)

$$
\begin{aligned}
&AEnv \coloneqq Var \to AVal \\
&\mathcal{M}(a) \coloneqq AEnv \times Kon \to \mathcal{P}(a \times AEnv \times Kon) \\
&\text{atom} : Atom \to \mathcal{M}(Exp) \\
&\text{atom}(a) \coloneqq \underline{\text{do}} \\
&\quad v \leftarrow \mathcal{A}[\![a]\!] \\
&\quad fr \leftarrow \text{pop-Kon} \\
&\quad \underline{\text{case}}\ fr\ \underline{\text{of}} \\
&\qquad [\square\ op\ e] \to \underline{\text{do}} \\
&\qquad\quad \text{push-Kon}([v\ op\ \square]) \\
&\qquad\quad \underline{\text{return}}(e) \\
&\qquad [v'\ op\ \square] \to \underline{\text{return}}(\delta(op, v', v)) \\
&\qquad [\square\ \ e] \to \underline{\text{do}} \\
&\qquad\quad \text{push-Kon}([v\ \ \square]) \\
&\qquad\quad \underline{\text{return}}(e) \\
&\qquad [v'\ \ \square] \to \underline{\text{do}} \\
&\qquad\quad \langle \boldsymbol{\lambda}(x) \to e, \rho' \rangle \leftarrow \text{branch-on-set}(\text{clo-E}(v')) \\
&\qquad\quad \text{put-Env}(\rho'[x \mapsto v]) \\
&\qquad\quad \underline{\text{return}}(e) \\
&\qquad [\textbf{if0}(\square)\{e_1\}\{e_2\}] \to \underline{\text{do}} \\
&\qquad\quad b \leftarrow \text{branch-on-set}(\text{if0-E}(v)) \\
&\qquad\quad \underline{\text{if}}\ v \stackrel{?}{=} 0\ \underline{\text{then}}\ \underline{\text{return}}(e_1)\ \underline{\text{else}}\ \underline{\text{return}}(e_2)
\end{aligned}
$$

Figure 4: λ-IF monadic interpreter with abstract domain exposed.