Modular Static Analysis: Orthogonal Design, Correct by Construction

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

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.

Analyses in our framework are least fixed point computations over a collecting semantics. A collecting semantics for **λ-IF** is defined as:

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

[Copyright notice will appear here once 'preprint' option is removed.]

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

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

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.

1.2 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 section.

The interpreter will use two types monadic effects: state and partiality. The operations for state effects are get and put, and the operation for the partiality effect is $\bot$. For reference: monadic, state and partiality operations have the following types:

$$
\begin{aligned}
&\mathit{return} : \forall \alpha, \alpha \to \mathcal{M}(\alpha) \\
&\mathit{extend} : \forall \alpha\beta, (\alpha \to \mathcal{M}(\beta)) \to (\mathcal{M}(\alpha) \to \mathcal{M}(\beta)) \\
&\quad \mathit{get} : \mathcal{M}(s) \\
&\quad \mathit{put} : s \to \mathcal{M}(1) \\
&\quad \bot : \forall \alpha, \mathcal{M}(\alpha)
\end{aligned}
$$

We construct a monad $\mathcal{M}$ for our interpreter which combines state effects for Env and Kon with a partiality effect. The type $\mathcal{M}$ is just a simple type:

$$
\mathcal{M}(\alpha) \coloneqq \mathit{Env} \times \mathit{Kon} \to (\alpha \times \mathit{Env} \times \mathit{Kon})_\bot
$$

We factor the definition of the interpreter into two parts: one for atomic and one for compound expressions. Atomic expressions return to the top frame of the stack:

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

Our interpreter uses helper functions $\mathcal{A}[\![\_]\!]$, push-Kon, pop-Kon, δ, $\downarrow_{\text{Clo}}$, and $\uparrow_\bot$. The function $\mathcal{A}[\![\_]\!]$ is the denotation function which uses the monadic state effect for environments:

$$
\begin{aligned}
&\mathcal{A}[\![\_]\!] : \mathit{Atom} \to \mathcal{M}(\mathit{Val}) \\
&\mathcal{A}[\![x]\!] \coloneqq \underline{\text{do}} \\
&\quad \rho \leftarrow \text{get-Env} \\
&\quad \uparrow_\bot(\rho(x)) \\
&\mathcal{A}[\![i]\!] \coloneqq \text{return}(i) \\
&\mathcal{A}[\![\boldsymbol{\lambda}(x) \to e]\!] \coloneqq \underline{\text{do}} \\
&\quad \rho \leftarrow \text{get-Env} \\
&\quad \text{return}(\langle \boldsymbol{\lambda}(x) \to e, \rho \rangle)
\end{aligned}
$$

Functions push-Kon and pop-Kon interact with effects for both continuation state and partiality:

$$
\begin{aligned}
&\text{push-Kon} : \mathit{Frame} \to \mathcal{M}(1) \\
&\text{push-Kon}\ \mathit{fr} \coloneqq \underline{\text{do}} \\
&\quad \overline{\mathit{fr}} \leftarrow \text{get-Kon} \\
&\quad \text{put-Kon}(\mathit{fr} :: \overline{\mathit{fr}}) \\
&\text{pop-Kon} : \mathcal{M}(\mathit{Frame} \times \mathit{Frame}^*) \\
&\text{pop-Kon} \coloneqq \underline{\text{do}} \\
&\quad \overline{\mathit{fr}} \leftarrow \text{get-Kon} \\
&\quad \underline{\text{case}}\ \overline{\mathit{fr}}\ \underline{\text{of}} \\
&\qquad \bullet \to \bot \\
&\qquad \mathit{fr} :: \overline{\mathit{fr'}} \to \underline{\text{do}} \\
&\qquad\quad \text{put-Kon}(\overline{\mathit{fr'}}) \\
&\qquad\quad \text{return}(\mathit{fr})
\end{aligned}
$$

The function δ gives the expected denotation to operators:

$$
\begin{aligned}
&\delta : \mathit{Op} \times \mathit{Val} \times \mathit{Val} \to \mathit{Val}_\bot \\
&\delta(+, i_1, i_2) \coloneqq i_1 + i_2 \\
&\delta(-, i_1, i_2) \coloneqq i_1 - i_2 \\
&\delta(\_, \_, \_) = \bot
\end{aligned}
$$

The function $\downarrow_{\text{Clo}}$ is a function from types which include closure values to partial closures.

$$
\begin{aligned}
&\downarrow_{\text{Clo}} : \mathit{Val} \to \mathit{Clo}_\bot \\
&\downarrow_{\text{Clo}}(\mathit{clo}) \coloneqq \mathit{clo} \\
&\downarrow_{\text{Clo}}(\_) \coloneqq \bot
\end{aligned}
$$

The function $\uparrow_\bot$ lifts partial values to monadic values, mapping $\bot$ to $\bot$.

$$
\begin{aligned}
&\uparrow_\bot : \forall \alpha, \alpha_\bot \to \mathcal{M}(\alpha) \\
&\uparrow_\bot(\bot) \coloneqq \bot \\
&\uparrow_\bot(a) \coloneqq \text{return}(a)
\end{aligned}
$$

Our small-step interpreter is defined by dispatching to atom or pushing continuation frames for compound expressions.

$$
\begin{aligned}
&\text{step} : \mathit{Exp} \to \mathcal{M}(\mathit{Exp}) \\
&\text{step}(a) \coloneqq \text{atom}(a) \\
&\text{step}(e_1\ \mathit{op}\ e_2) \coloneqq \underline{\text{do}} \\
&\quad \text{push-Kon}([\square\ \mathit{op}\ e_2]) \\
&\quad \text{return}(e_1) \\
&\text{step}(e_1\ \ e_2) \coloneqq \underline{\text{do}} \\
&\quad \text{push-Kon}([\square\ \ e_2]) \\
&\quad \text{return}(e_1) \\
&\text{step}(\textbf{if0}(e_1)\{e_2\}\{e_3\}) \coloneqq \underline{\text{do}} \\
&\quad \text{push-Kon}([\textbf{if0}(\square)\{e_2\}\{e_3\}]) \\
&\quad \text{return}(e_1)
\end{aligned}
$$

Correctness and Execution As before, we must construct a collecting semantics in order to define an executable analysis. To do this, we must relate our small-step interpreter to a state machine transition function. That is, we must relate our interpreter of type

$Exp \to \mathcal{M}(Exp)$ to a state transition of type $\Sigma \to \Sigma$ for *some* Σ.

For this interpreter the state space Σ is defined $\Sigma := (Exp \times Env \times Kon)_\perp$. To establish the correctness of Σ we construct an isomorphism between $Exp \to \mathcal{M}(Exp)$ and $\Sigma \to \Sigma$. We use the standard isomorphisim between Kleisli morphisms $\alpha \to \mathcal{M}(\beta)$ and $\mathcal{M}(\alpha) \to \mathcal{M}(\beta)$ where $\mathcal{M}$ is the partiality monad $(\_)_\perp$. To execute the analysis we transport the interpreter through the isomorphism and find the least fixed point of the collecting semantics.

1.3 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 $\widehat{Val}$ behind an abstract interface:

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

This abstraction forces us to use an abstract environment, mapping variables to abstract values:

$$\widehat{Env} := Var \rightharpoonup \widehat{Val}$$

and re-implement $\mathcal{A}[\![\_]\!]$:

$$
\begin{aligned}
&\mathcal{A}[\![\_]\!] : Atom \to \mathcal{M}(\widehat{Val}) \\
&\mathcal{A}[\![x]\!] := \underline{\text{do}} \\
&\quad \rho \leftarrow \text{get-Env} \\
&\quad \uparrow_\perp(\rho(x)) \\
&\mathcal{A}[\![i]\!] := \text{return}(\widehat{\text{int-I}}(i)) \\
&\mathcal{A}[\![\boldsymbol{\lambda}(x) \to e]\!] := \underline{\text{do}} \\
&\quad \rho \leftarrow \text{get-Env} \\
&\quad \text{return}(\widehat{\text{clo-I}}(\langle \boldsymbol{\lambda}(x) \to e, \rho \rangle))
\end{aligned}
$$

Before implementing atom we must reconsider our monadic effects. Because we have abstracted values, our interpreter must consider *sets* of possible states resulting from conditionals and function calls. We introduce this nondeterminism as an effect in the monad, taking the place of partiality (which was just nondeterminism with zero or one branches). Nondeterminism has two operators, $\perp$ as we've seen before, and $\langle + \rangle$ which branches to both left and right monadic computation. The effects for our monadic type are now:

$$
\begin{aligned}
&return : \forall \alpha, \alpha \to \mathcal{M}(\alpha) \\
&extend : \forall \alpha \beta, (\alpha \to \mathcal{M}(\beta)) \to (\mathcal{M}(\alpha) \to \mathcal{M}(\beta)) \\
&\quad get : \mathcal{M}(s) \\
&\quad put : s \to \mathcal{M}(1) \\
&\quad\ \perp : \forall \alpha, \mathcal{M}(\alpha) \\
&\quad \langle + \rangle : \forall \alpha, \mathcal{M}(\alpha) \to \mathcal{M}(\alpha) \to \mathcal{M}(\alpha)
\end{aligned}
$$

and to support these effects we redefine $\mathcal{M}$ to be:

$$\mathcal{M}(\alpha) := \widehat{Env} \times Kon \to \mathcal{P}(\alpha \times \widehat{Env} \times Kon)$$

Our definition for atom now uses nondeterminism effects:

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

Helper function $\uparrow_{\mathcal{P}}$ is the analog of $\uparrow_\perp$ for lifting sets into monads with nondeterminism:

$$
\begin{aligned}
&\uparrow_{\mathcal{P}} : \forall \alpha, \mathcal{P}(\alpha) \to \mathcal{M}(\alpha) \\
&\uparrow_{\mathcal{P}}(\{\}) := \perp \\
&\uparrow_{\mathcal{P}}(\{a_1 \ldots a\}) := \text{return}(a_1)\ \langle + \rangle\ \ldots\ \langle + \rangle\ \text{return}(a)
\end{aligned}
$$

Correctness and Execution To construct a collecting semantics we again construct an isomorphism between $Exp \to \mathcal{M}(Exp)$ and Σ where $\Sigma := \mathcal{P}(Exp \times \widehat{Env} \times Kon)$. This isomorphism is again the standard one between Kleisli morphisms $\alpha \to \mathcal{M}(\beta)$ and $\mathcal{M}(\alpha) \to \mathcal{M}(\beta)$ where $\mathcal{M}$ is the powerset monad $\mathcal{P}$.

Now that we have a parameter to the analysis, $\widehat{Val}$, we can discuss what an analysis is and what it means for it to be correct. An analysis is some instantiation of $\widehat{Val}$ to $\widehat{Val}_1$ where $Val \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \widehat{Val}_1$. Furthermore, each operation in the $\widehat{Val}$ interface must be an abstraction of concrete versions, namely int-I $\sqsubseteq \gamma(\widehat{\text{int-I}}_1)$, if0-E $\sqsubseteq \gamma(\widehat{\text{if0-E}}_1)$, etc. This local argument about $\widehat{Val}_1$, coupled with the monotonicity of step and the isomorphism between $Exp \to \mathcal{M}(Exp)$ and $\Sigma \to \Sigma$, gives us an end to end correctness argument for our executable abstract interpreter. At this point our analysis is still not yet decidable; the abstract state space is still infinite.

1.4 Exposing Context Sensitivity

To expose context sensitivity (also called call-site sensitivity) we instantiate the AAM approach of cutting recursion in the state space to our framework. This conceptually takes place in two steps. First we introduce indirection between values and environments through a store and a type of addresses. Second we abstract the choice of address to achieve an analysis.

We introduce abstract time and addresses behind an interface:

$$\begin{aligned}
&\widehat{Addr} \in Set \\
&\widehat{Time} \in Set \\
&\widehat{alloc} : \widehat{Time} \to \widehat{Addr} \\
&\widehat{tick} : (Exp \times Kon \times \widehat{Time}) -> \widehat{Time}
\end{aligned}$$

which alters the definition of the monad to contain new state components:

$$\begin{aligned}
\widehat{Env} &:= Var \rightharpoonup Addr \\
\widehat{Store} &:= Addr \to \widehat{Val} \\
\mathcal{M}(\alpha) &:= \widehat{Env} \times Kon \times \widehat{Time} \times \widehat{Store} \\
&\quad \to \mathcal{P}(\alpha \times \widehat{Env} \times Kon \times \widehat{Time} \times \widehat{Store})
\end{aligned}$$

[LEFT OFF HERE]

1.5 Exposing Flow Sensitivity

Novel in this work is the ability to expose flow sensitivity as an orthogonal parameter to a static analysis. To do this we exploit both the monadic abstraction and our correctness framework which relates monadic actions to state machine transitions. *Our insight is that both computational and correctness components of flow-sensitivity can be obtain by varying the monad for the interpreter alone.*

The anlyses we have constructed thus far are path sensitive, meaning they will follow branching control flow precisely. We can see that the analysis is path sensitive by inspecting the state space Σ that it induces: $\Sigma := \mathcal{P}(Exp \times \widehat{Env} \times Kon \times \widehat{Time} \times \widehat{Store})$. A flow insensitive analysis pulls the store out of the powerset to a single global store: $\Sigma := (\mathcal{P}(Exp \times \widehat{Env} \times Kon \times \widehat{Time}) \times \widehat{Store})$. A flow sensitive path insensitive analysis shares the state space of path sensitivity but employs store widening to keep the state space explosion under control. In this section we will show how variations in the monad can recover all three analysis variants: path sensitive, flow sensitive, and flow insensitive.

1.6 Intentional Optimizations

1.6.1 Abstract Garbage Collection

1.6.2 Flat Closures

2. Summary

Our final interpreter enjoys the following properties:

- Abstract Domain can be altered independent of other features.
- Flow Sensitivity can be altered independent of other features.
- Context Sensitivity (and its granularity) can be altered indepndend of other features.
- Intensional optimizations can be implemented once and integrate seamlessly with any of the above choices.
- New state space components and their widening properties can be added seamlessly using compositional building blocks.
- Interpreters are automatically related to a collecting semantics, enabling a straight forward least-fixed-point implementation strategy.
- All analyses are correct by construction with no additional proof burdon beyond interpreter monotonicity and correctness of orthoganal parameters piecewise.