Abstract Control in Static Analysis

David Darais

September 30, 2014

Abstract

The field of static analysis offers many techniques for automatically discovering facts about programs. For example, a static analysis might be used to verify that a program is memory safe, or that it obeys a privacy policy. Techniques based on Abstract Interpretation (AI) allow the analysis designer to vary some properties of an analysis. However, features like context, object, and flow sensitivity are often hard-coded into an analysis based on classic AI. Abstracting Abstract Machines (AAM) is one design methodology for analysis which exposes more of these features as tuning knobs. Unfortunately, AAM lacks reusable tools for building analyses, and not all analysis features fit nicely into the framework.

We propose a new compositional framework for designing analyses which support a tuning knob missing from AAM: abstract control. We achieve compositionality by refactoring AAM into monad transformers, each of which we relate back to abstract state machines piecewise. When put together, these monad transformers build analyses which are correct by construction. Furthermore, the monadic abstraction allows us to recover a new tuning knob which we call abstract control. Abstract control allows one to adjust the flow and path sensitivity of an analysis independent of other design choices.

Contents

1 Introduction

Static analysis has many applications. A software developer might use an analysis to find bugs in a program. A software merchant might use an analysis to detect malware or other security vulnerabilities in software hosted by their app store. A manufacturer might use an analysis to establish high assurance in the reliability of control software for a car or airplane. Each of these situations requires a different type of analysis. To meet the needs of a given problem, an analysis designer has two options: retrofit an existing analysis or design one from scratch. Retrofitting an existing analysis can be insufficient because the tension between precision and performance is often too high. Getting an analysis right might require more modification than an existing analysis was designed to support. Because of this, complex analyses often need to be designed from scratch, and analysis designer need tools to help them with this process.

Designing a static analysis is hard. Designing *the right* static analysis is even harder. One possible process for arriving at the right static analysis is:

1. Describe the semantics of your language.
2. Design an analysis which you hope has right properties.
3. Exercise the analysis.
4. If the analysis is too slow or imprecise: repeat from 2.

To get a complex analysis right this process must be repeated multiple times. Iterating quickly between the design and exercise of an analysis is the key to getting the analysis you wanted without too much unnecessary effort.

One methodology which aims to ease the design process for static analysis is the Abstracting Abstract Machines (AAM) framework[15]. AAM separates two design steps which are traditionally tightly coupled: describing features of interest and abstracting those features. Furthermore, AAM shows that the abstraction step is systematic, freeing the analysis designer to spend effort in designing features of interest. However, AAM doesn't provide reusable tools for constructing analyses. The analysis designer must describe their state machine transition function from scratch. Additionally, some analysis features like flow and path sensitivity don't fit nicely into the AAM framework.

We propose a new compositional framework for designing analyses which supports a tuning knob missing from AAM: abstract control. To achieve this we refactor AAM into monad transformers. Using monad transformers, the analysis designer can easily add and remove features from their semantics independently of other features. The monadic abstraction additionally exposes a new tuning knob for analyses which we call abstract control. Abstract control allows one to adjust the flow and path sensitivity of an analysis independent of other design choices.

To justify the soundness of the approach, we show that the monad transformers we use are also Galois connection transformers. This means that any combination of our monad transformers will yield a sound approximation of a state machine semantics.

Our contributions in this work are:

1. A monadic formulation of AAM which exposes abstract control as a tuning knob.
2. A compositional analysis *implementation* framework for building abstract state machines using monad transformers.
3. A compositional analysis *proof* framework for justifying the correctness of abstract state machines using Galois connection transformers.
4. A new monad transformer for nondeterminism which serves as our gateway to recovering abstract control as a tuning knob.

The rest of the paper is structured as follows: Section 2 covers necessary background material in semantics, analysis, and the CPS language which we use for examples. Section 3 demonstrates a monadic variant of AAM which supports defining compositional interpreters for analysis. Section 4 discusses abstract control and how we exploit the monadic abstraction to expose a tuning knob for it. Section 5 gives two examples of analysis optimizations and how they adapt seamlessly to variations in abstract control. Section 6 discusses the correctness framework and what key lemmas are involved. Section 10 shows proof steps in more detail for the arguments made in section 6. Sections 7 and 8 discuss relate and future work, and section 9 concludes.

2 Background

In this section we discuss background material potentially useful for understanding our compositional analysis framework. This section provides merely a sketch these con-

cepts, and does not motivate them directly in the context of our work. Some readers may find it useful to skip ahead to section 3 and chase backlinks as they are needed and motivated. Or some may prefer to proceed directly with this section, consuming the document in a more bottom-up fashion. For these readers, we have done our best to keep this section brief.

2.1 Small-Step Semantics

We use the small-step operational approach to semantics pioneered by independently by Plotkin[12] and Felleisen[4]. In small-step operational semantics, expressions in a language are given meaning by mapping them to state machines. Formally, a small-step semantics for a language L is given by:

- A state machine with configuration states Σ.
- A transition relation $\sigma \mapsto \sigma'$ for $(\sigma, \sigma' : \Sigma)$.
- An embedding function $\eta : L \to \Sigma$.

The meaning of an expression e is then given by a final configuration σ reachable by $\eta(e)$ through $\mapsto$.

Small-step semantics contrast to previously developed denotational approaches. A denotational semantics for a language L is given by a denotation function $[\![\_]\!]$ mapping terms e to objects in a mathematical space. This denotation function can quickly demand non-trivial mathematical machinery for even very simple languages. Small-step methods side-step the need for such machinery at the cost of potentially useful mathematical structure. Many useful analysis properties can be states as properties of execution states of an abstract machine. Small-step semantics therefore serve as a nice middle-ground marrying formality and simplicity.

2.2 Continuation Passing Style

We use CPS-IF as an example language to demonstrate our framework. CPS-IF is a variant of continuation passing style λ-calculus extended with conditionals and branching. We use λ-calculus because it is the simplest language with support for higher-order control flow. λ-calculus can serve as the core target language for larger languages which appear in practice. We use CPS because the state machine for the semantics of CPS requires one less component and it simplifies our presentation. All lambda calculus terms can be translated to CPS terms through CPS-conversion, so there is no loss in language expressivity. We add conditionals to help exercise analyses which track branching structure in different ways.

We define the CPS-IF language as follows:

$$
\begin{aligned}
x, y, k &: Var ::= ...variables... \\
b, i, l &: Lit ::= \mathbb{Z} \cup \mathbb{B} \\
f, g &: Lam ::= \underline{\lambda}(x) \to e \mid \underline{\lambda}(x, k) \to e \\
a &: Atom ::= x \mid l \mid f \mid op(a) \\
op &: Op ::= \underline{\text{add1}} \mid \underline{\text{sub1}} \mid \underline{\text{gez}} \\
e &: Exp ::= \underline{\text{if}}(a)\{e\}\{e\} \mid a(a) \mid a(a, a) \mid \underline{\text{halt}}(a)
\end{aligned}
$$

CPS-IF supports integer and boolean literals, addition and subtraction by 1, and testing if an integer is greater than or equal to zero. The essence of CPS is in the syntactic restriction on application forms $a(a)$ and $a(a, a)$. Because a cannot be a nested expression, evaluating an Exp requires no evaluation stack.

The state machine Σ for CPS-IF is defined as:

$$
\begin{aligned}
\ell &: Addr := \mathbb{Z} \\
\rho &: Env := Var \to Addr \\
\theta &: Store := Addr \to Val \\
v &: Val ::= l \mid \langle f, \rho \rangle \\
\sigma &: \Sigma := Exp \times Env \times Store
\end{aligned}
$$

Environments map variables to addresses, and the store maps addresses to values. Values contain closures, which are lambda expressions paired with their closure environment.

The small-step semantics for CPS-IF are defined in two parts. First we define a *denotation function* $\mathscr{A}$ for $Atom$ expressions.

$$
\begin{aligned}
\mathscr{A} &: Env \times Store \times Atom \rightharpoonup Val \\
\mathscr{A}(\rho, \theta, x) &:= \theta(\rho(x)) \\
\mathscr{A}(\rho, \theta, l) &:= \{l\} \\
\mathscr{A}(\rho, \theta, \underline{\text{add1}}(a)) &:= \mathscr{A}(\rho, \theta, a) + 1 \\
\mathscr{A}(\rho, \theta, \underline{\text{sub1}}(a)) &:= \mathscr{A}(\rho, \theta, a) - 1 \\
\mathscr{A}(\rho, \theta, \underline{\text{gez}}(a)) &:= \{True \text{ if } \mathscr{A}(\rho, \theta, a) >= 0, False \text{ otherwise}\} \\
\mathscr{A}(\rho, \theta, \underline{\lambda}(x) \to e) &:= \langle \underline{\lambda}(x) \to e, \rho \rangle \\
\mathscr{A}(\rho, \theta, \underline{\lambda}(x)(k) \to e) &:= \langle \underline{\lambda}(x)(k) \to e, \rho \rangle
\end{aligned}
$$

Second we define a *step relation* (as a function) $\mathscr{E}$ for Exp expressions.

$$\mathscr{E} : Exp \times Env \times Store \rightharpoonup Exp \times Env \times Store$$

$$\mathscr{E}(\underline{\text{if}}(a)\{e_1\}\{e_2\}, \rho, \theta) := \{(e_1, \rho, \theta) \text{ if } \mathscr{A}(\rho, \theta, a) = True, (e_2, \rho, \theta) \text{ if } \mathscr{A}(\rho, \theta, a) = False\}$$

$$\mathscr{E}(a_1(a_2, a_3), \rho, \theta) := (e, \rho'', \theta'')$$

where

$$\begin{aligned}
\langle \underline{\lambda}(x)(k) \to e, \rho' \rangle &= \mathscr{A}(a_1, \rho, \theta) \\
v_2 &= \mathscr{A}(a_2, \rho, \theta) \\
v_3 &= \mathscr{A}(a_3, \rho, \theta) \\
\ell_2 &= fresh(\theta) \\
\theta' &= \theta[\ell_2 \mapsto v_2] \\
\ell_3 &= fresh(\theta') \\
\theta'' &= \theta'[\ell_3 \mapsto v_3] \\
\rho'' &= \rho'[x \mapsto \ell_2][k \mapsto \ell_3]
\end{aligned}$$

$$\mathscr{E}(\underline{\text{halt}}(a), \rho, \theta) := (\underline{\text{halt}}(a), \rho, \theta)$$

2.3 Abstract Interpretation

Abstract Iterpretation (AI) is a formal framework for program analysis pioneered by Cousot and Cousot[2]. In the setting of abstract interpretation, a program analysis is just an alternate semantics over an abstract domain. To distinguish the original semantics from the alternate semantics we call the original semantics the *concrete* semantics. The alternate semantics is called the *abstract* semantics[1].

Consider a concrete state machine Σ and a state $(\sigma : \Sigma)$. In the AI framework, an analysis for σ is:

- An abstract state machine $\widehat{\Sigma}$.
- A relationship between Σ and $\widehat{\Sigma}$ that defines when some $(\sigma' : \Sigma)$ and $(\widehat{\sigma}' : \widehat{\Sigma})$ are related. This often takes the form of projecting $\widehat{\Sigma}$ to $\mathscr{P}(\Sigma)$ and using the subset relation.
- An abstract version of σ: $(\widehat{\sigma} : \widehat{\Sigma})$.
- An abstract version of the $\mapsto$ relation: $(\ \widehat{\mapsto}\ : \widehat{\Sigma} \times \widehat{\Sigma} \to Prop)$
- A method to explore every state reachable by $\widehat{\sigma}$ under $\ \widehat{\mapsto}$. This often requires $\widehat{\Sigma}$ to be finite.

[1] This terminology quickly becomes confusing when we construct an abstract machine for both semantics. This yields a concrete abstract machine and an abstract abstract machine. This linguistic pun is exploited mercilessly in the work of Might and Van Horn as they "abstract" abstract abstract machines. We simplify the matter by calling them abstract state machines and concrete state machines.

By convention, we put hats ($\hat{}$) on things to name their abstract cousins. The overall approach is summarized in the following picture:

$$\begin{array}{ccc} (\sigma : \Sigma) & \mapsto & (\sigma' : \Sigma) \\ R \Big| & & R \Big| \\ (\hat{\sigma} : \hat{\Sigma}) & \hat{\mapsto} & (\hat{\sigma}' : \hat{\Sigma}) \end{array}$$

This picture takes place at some point in the analysis process. σ is the current state (possibly the result of running the original program for a little while). $\hat{\sigma}$ is a valid abstraction of σ, where this validity is expressed by some relation R holding between σ and $\hat{\sigma}$. σ' is some next state of execution of σ, and likewise for $\hat{\sigma}/\hat{\sigma}'$. In order to be a correct analysis, it must be guaranteed that σ' and $\hat{\sigma}'$ will be related. The logical structure of the picture, which states the correctness of the analysis $\hat{\mapsto}$, is given by:

$$\forall(\sigma, \sigma' : \Sigma)(\hat{\sigma}, \hat{\sigma}' : \hat{\Sigma}), (\sigma R \hat{\sigma}) \wedge (\sigma \mapsto \sigma') \wedge (\hat{\sigma} \hat{\mapsto} \hat{\sigma}') \Rightarrow (\sigma' R \hat{\sigma}')$$

The meaning of the analysis is entirely subject to the relation R which relates the abstract to the concrete.

2.4 Galois Connections

The AI setting can be elegantly simplified and enriched through the use of *Galois connections*. Galois connections serve as a unifying framework for establishing the "relationship between Σ and $\hat{\Sigma}$" mentioned in the previous section.

A Galois connection between two posets (sets with a partial order) Σ and $\hat{\Sigma}$ is notated $\Sigma \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{\Sigma}$ and contains:

- $(\alpha : \Sigma \to \hat{\Sigma})$ where α is monotonic
- $(\gamma : \hat{\Sigma} \to \Sigma)$ where γ is monotonic
- A proof that $(\gamma \circ \alpha)$ is expansive: $\forall(x : \Sigma), x \sqsubseteq \gamma(\alpha(x))$
- A proof that $(\alpha \circ \gamma)$ is contractive: $\forall(y : \hat{\Sigma}), \alpha(\gamma(y)) \sqsubseteq y$

The last two properties can be succinctly stated as $(\alpha \circ \gamma \sqsubseteq id \sqsubseteq \gamma \circ \alpha)$[2]. Equivalent to all four properties is the property $x \sqsubseteq \gamma(y) \Leftrightarrow \alpha(x) \sqsubseteq y$.

A Galois connection $\Sigma \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{\Sigma}$ can be read "$\hat{\Sigma}$ is an abstraction of Σ" and $x \sqsubseteq y$ can be read "x is more precise than y". The expansive property corresponds to *soundness*, and the contractive property corresponds to *tightness*. A sound analysis gives results you can trust. A tight analysis promises to be the "best" analysis possible.

[2]This uses the usual monotonicity relation $f \sqsubseteq g \Leftrightarrow (x \sqsubseteq y \Rightarrow f(x) \sqsubseteq g(y))$ for the function space.

Example: Given a Galois connection $\Sigma \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \widehat{\Sigma}$, there exists a Galois connection $(\Sigma \xrightarrow{mon} \Sigma) \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} (\widehat{\Sigma} \xrightarrow{mon} \widehat{\Sigma})$ where:

$$\alpha'(f : \Sigma \xrightarrow{mon} \Sigma) \coloneqq \alpha \circ f \circ \gamma$$
$$\gamma'(g : \widehat{\Sigma} \xrightarrow{mon} \widehat{\Sigma}) \coloneqq \gamma \circ g \circ \alpha$$

Example: The language $(\mathscr{P}(\mathbb{Z}), +, *)$ forms a Galois connection with the language $(\mathscr{P}(\{EVEN, ODD\}), \wedge, \vee)$ where:

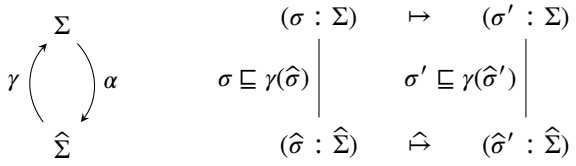
$$\alpha(zs : \mathscr{P}(\mathbb{Z})) \coloneqq \{EVEN \mid \exists z \in zs \wedge Even(z)\} \cup \{ODD \mid \exists z \in zs \wedge Odd(z)\}$$
$$\gamma(ts : \mathscr{P}(\{EVEN, ODD\})) \coloneqq \{z \in \mathbb{Z} \mid EVEN \in ts \wedge Even(z)\} \cup \{z \in \mathbb{Z} \mid ODD \in ts \wedge Odd(z)\}$$

Galois connections simplify the AI framework by using $x \sqsubseteq \gamma(y)$ or (equivalently) $\alpha(x) \sqsubseteq y$ as the relation (xRy). Galois connections are a natural and general way of placing partial orders *on sets themselves*. $x \sqsubseteq \gamma(y)$ can be seen as a heterogenous extension of $x \sqsubseteq y$ when $(x : A)$ and $(y : B)$ live in different sets. One can also think of Galois connections as something like an isomorphism, but with a weaker round-trip property. (An isomorphism would require $\alpha \circ \gamma = id = \gamma \circ \alpha$.)

Using Galois connections, the AI framework introduced in the previous section can be re-stated. In the AI framework, an analysis for σ is:

- An abstract language $\widehat{\Sigma}$.
- A Galois connection $\Sigma \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \widehat{\Sigma}$.
- An abstract version of the $\mapsto$ relation: $(\widehat{\mapsto} : \widehat{\Sigma} \times \widehat{\Sigma} \to Prop)$
- A way to explore every state reachable by $\widehat{\sigma}$ under $\widehat{\mapsto}$. This often requires $\widehat{\Sigma}$ to be finite.

The overall approach is summarized in the following picture:



As before, the logical structure of the picture, which states the correctness of the analysis $\widehat{\mapsto}$, is given by:

$$\forall(\sigma, \sigma' : \Sigma)(\widehat{\sigma}, \widehat{\sigma}' : \widehat{\Sigma}), (\sigma \sqsubseteq \gamma(\widehat{\sigma})) \wedge (\sigma \mapsto \sigma') \wedge (\widehat{\sigma} \mathrel{\widehat{\mapsto}} \widehat{\sigma}') \Rightarrow (\sigma' \sqsubseteq \gamma(\widehat{\sigma}'))$$

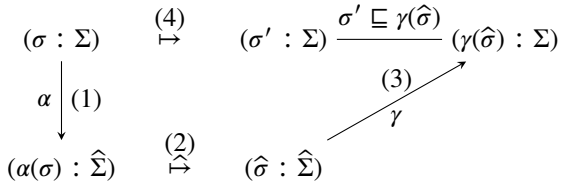
The statement of this property can be simplified further as $\mapsto \sqsubseteq \gamma(\ \widehat{\mapsto}\ )$. This uses the definition of Galois connections for function spaces shown in a previous example.

Using a Galois connection $\Sigma \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \widehat{\Sigma}$ and abstract step function $\widehat{\mapsto} \sqsubseteq \gamma(\mapsto)$, an algorithm for analysis can be stated:

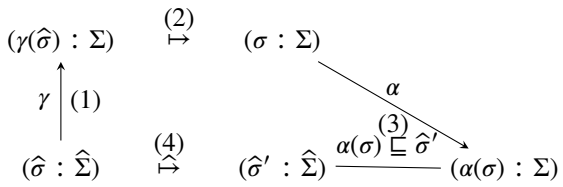
1. Translate $(\sigma : \Sigma)$ to $(\alpha(\sigma) : \widehat{\Sigma})$
2. Explore all abstract states $\widehat{\sigma}'$ states reachable from $\alpha(\sigma)$.

All reachable concrete states are then summarized by projecting each $\gamma(\widehat{\sigma}')$ back to Σ.

The Galois connection framework simultaneously justifies the *soundness* and *tightness* of this method. Soundness tells us that if we (1) abstract, (2) take an abstract step, and then (3) concretize, then the result is an approximation of (4) taking a concrete step. Soundness tells us that we can trust the results of the analysis.



Tightness tells us that if we (1) concretize, (2) take a concrete step, and then (3) abstract, then the result must be just as precise as (4) taking an abstract step. Tightness tells us that our abstract step function isn't losing precision unnecessarily, and that our abstract step $\widehat{\mapsto}$ is provably as precise as possible.



The above diagrams compose to state similar properties about $\mapsto *$ and $\widehat{\mapsto} *$.

2.5 Control Flow Analysis

Control flow analysis is a class of analysis which is particularly important for higher-order languages. In non-higher-order languages, it can be useful to distinguish control-flow from data-flow. The control flow of the program is the graph of what functions are called from where. The data flow of a program is the mapping of which values flow to which variables. Traditionally one performs a control-flow analysis first to find out which functions are called, and a data-flow analysis second using the results of the control flow analysis. In higher-order languages, data-flow and control-flow are tightly coupled. Before you can tell which functions will be called, you need to know how values will flow through the program.

We note that the distinction between "higher-order" languages and "non-higher-order" languages is a red herring. Functions can be passed as values in C using function pointers. Object-oriented languages enjoy a similar circularity between control and data flow from dynamic method dispatch. You can read these statements as saying "all languages are higher order, and thus all languages need higher order control flow analysis". Or you can read them as saying "all languages are higher order, and we've done just fine without higher oder control flow analysis so far". The distinction in practice comes from the properties you are looking for and the code that you end up analysing.

We build on the tradition of Control Flow Analysis (CFA) pioneered by Shivers[13]. In this tradition, the result of a control flow analysis is closer to what one would think of as a data-flow analysis. CFA is nothing more than getting data-flow right in a higher-order setting. 0CFA, the most basic control flow analysis, merely computes the set of values which might flow to a particular variable. kCFA is a family of context sensitive extensions to 0CFA which separates analysis results for functions based on call site. We use 0CFA and kCFA as the example analyses in this paper.

2.6 Monads

Some find it useful to review the concept of functor before learning about monads. We adopt this approach for our brief explanation of monads.

Functors A type ($F : Set \to Set$) is called a *functor* if one can:

- Define $map : \forall(A, B : Set), (A \to B) \to (F(A) \to F(B))$
- Prove $map(id) = id$
- Prove $map(g \circ f) = map(g) \circ map(f)$

Example. Lists are functors, where *map* is defined:

$$\begin{aligned} map(f)(xs) := & \underline{\text{case}}(xs) : \\ & Nil \to Nil \\ & x :: xs' \to f(x) :: map(f)(xs') \end{aligned}$$

Example. $map(isEven)([1,2,3,4]) = [False, True, False, True]$

Monads A type ($\mathscr{M} : Set \to Set$) is called a *monad* if one can:

- Define $extend : \forall(A, B : Set), (A \to \mathscr{M}(B)) \to (\mathscr{M}(A) \to \mathscr{M}(B))$
- Prove $extend(return) = id$, called the left-unit law.
- Prove $extend(f)(return(x)) = f(x)$, called the right-unit law.
- Prove $extend(g)(extend(f)(x)) = extend(\lambda(y) \to extend(g)(f(y)))(x)$ called associativity.

The only computational difference from the definition of functor is in the first argument of extend. For monads, this argument is allowed to be a *monadic* ($A \to \mathscr{M}(B)$), rather than a pure function ($A \to B$).

Example. Lists are monads, where *extend* is defined:

$$\begin{aligned} extend(f)(xs) \coloneqq & \underline{\text{case}}(xs) : \\ & Nil \to Nil \\ & x :: xs' \to f(x) +\!\!+^3 extend(f)(xs') \end{aligned}$$

Example. Let $adjacent(x) \coloneqq [x-1, x+1]$. Then $extend(adjacent)([10, 100]) = [9, 11, 99, 101]$

Monad Notation One of the biggest benefits of monads is perhaps the most superficial: do-notation. Do-notation transforms imperative-looking code into uses of *extend*. This enables kind of "abstract semicolon", where the meaning of sequencing statements is give by an arbitrary implementation of *extend*.

Do-notation is simply a translation from:

$$\begin{aligned} & do \\ & \quad x \leftarrow get \\ & \quad y \leftarrow get \\ & \quad put(x + y) \end{aligned}$$

to

$$extend(\lambda(x) \to extend(\lambda(y) \to put(x + y)(get)))(get)$$

Leveraging even more notation, we can write $extend(f)(x)$ infix and backwards as $x \gg\!= f$, yielding the following equivalent translation:

$$\begin{aligned} & get \gg\!= \lambda(x) \to \\ & get \gg\!= \lambda(y) \to \\ & put(x + y) \end{aligned}$$

which looks closer to the original expression written in do-notation. For the rest of the paper we will write exclusively using do-notation and assume this lightweight translation to uses of *extend*.

2.7 Monadic Effects

In addition to *return* and *extend*, monads typically come with their own flavor of *monadic effect*. We use two monadic effects in this paper: state and nondeterminism.

State The state monadic effect models manipulation of a single cell of state with type $\mathcal{J}$ using operations *get* and *put*:

$$\begin{aligned} \mathit{get} &: \mathscr{M}(\mathcal{J}) \\ \mathit{put} &: \mathcal{J} \to \mathscr{M}(1) \end{aligned}$$

(We write 1 as the unit type, i.e. the singleton set.) *get* and *put* come with the following laws:

$$\begin{aligned} \mathit{put}(\mathcal{J}_1); \mathit{put}(\mathcal{J}_2) &= \mathit{put}(\mathcal{J}_2) \\ \mathit{put}(\mathcal{J}); \mathit{get} &= \mathit{return}(\mathcal{J}) \\ \mathcal{J} \leftarrow \mathit{get}; \mathit{put}(\mathcal{J}) &= \mathit{return}(\bullet) \\ \mathcal{J}_1 \leftarrow \mathit{get}; \mathcal{J}_2 \leftarrow \mathit{get}; k(\mathcal{J}_1, \mathcal{J}_2) &= \mathcal{J} \leftarrow \mathit{get}; k(\mathcal{J}, \mathcal{J}) \end{aligned}$$

Multiple cells of state can be modeled by stacking this effect multiple times.

Nondeterminism The nondeterminism monadic effect models branching control flow using a single operation $\langle + \rangle$:

$$\begin{aligned} \bot &: \forall (a : A), \mathscr{M}(A) \\ \langle + \rangle &: \forall (a : A), \mathscr{M}(A) \to \mathscr{M}(A) \to \mathscr{M}(A) \end{aligned}$$

$\langle + \rangle$ comes with the following laws:

$$\begin{aligned} \mathit{extend}(\_)(\bot) &= \bot \\ \mathit{extend}(\mathit{const}(\bot))(\_) &= \bot \\ \mathit{extend}(f)(x \ \langle + \rangle \ y) &= \mathit{extend}(f)(x) \ \langle + \rangle \ \mathit{extend}(f)(y) \end{aligned}$$

A null branch of a computation is written $\bot$ and branching with two possible values $v_1 v_2$ is written $\mathit{return}(v_1) \ \langle + \rangle \ \mathit{return}(v_2)$. Note that $\mathscr{M}$ having a nondeterminism effect is equivalent to $\mathscr{M}$ being a join-semilattice for every possible $\mathscr{M}(a)$.

2.8 Monad Transformers

Monad transformers are building blocks for constructing larger monads. Where a monad $\mathscr{M}$ will have type $\mathit{Set} \to \mathit{Set}$, a monad transformer $\mathscr{T}$ will have type $(\mathit{Set} \to \mathit{Set}) \to (\mathit{Set} \to \mathit{Set})$. Monad transformers are used to extend an existing monad to support another effect.

The three monads used in this work are the state monad, nondeterminism monad, and identity monad. The state monad is notated $\mathcal{S}(\mathcal{J})$ (carrying a single cell of type $\mathcal{J}$), nondeterminism $\mathcal{P}$, and identity ID. The transformer analogues for state and nondeterminism are notated $\mathcal{S}_T(\mathcal{J})(\mathscr{M})$ and $\mathcal{P}_T$ $(\mathscr{M})$ where $\mathscr{M}$ is the underlying monad. The transformer versions of both of these monads allow you to combine effects piecewise.

Example. $\mathcal{S}_T(\mathbb{Z})(\mathcal{P}_T\ (ID))$ and $\mathcal{P}_T\ (\mathcal{S}_T(\mathbb{Z})(ID))$ are both monads with state (where $\mathcal{s} = \mathbb{Z}$) and nondeterminism effects.

The non-transformer versions of state and nondeterminism monads can be defined directly, or simply as transformers on top of ID. The nondeterminism monad transformer we use in this work is new. The full definitions and proofs of all monad transformers used in this paper are given in section 10.

3 A Compositional Analysis Framework

A compositional analysis framework must support adding and removing features from the analysis without touching the entire implementation. We achieve such compositionality by building abstract interpreters using monad transformers. To demonstrating our compositional approach to analysis design, we first show a classic definition of 0CFA. Then, we will show how to design such an analysis using our framework. After porting 0CFA to our framework, we will show how to grow this analysis to include context sensitivity in a compositional fashion.

We will use the CPS-IF language introduced in section 2.2 as the target language for 0CFA. We repeat the definition of CPS-IF syntax here for convenience.

$$
\begin{aligned}
x, y, k &: Var ::= ...variables... \\
b, i, l &: Lit ::= \mathbb{Z} \cup \mathbb{B} \\
f, g &: Lam ::= \underline{\lambda}(x) \to e \mid \underline{\lambda}(x, k) \to e \\
a &: Atom ::= x \mid l \mid f \mid op(a) \\
op &: Op ::= \underline{\text{add1}} \mid \underline{\text{sub1}} \mid \underline{\text{gez}} \\
e &: Exp ::= \underline{\text{if}}(a)\{e\}\{e\} \mid a(a) \mid a(a, a) \mid \underline{\text{halt}}(a)
\end{aligned}
$$

3.1 Classic 0CFA

Our abstract semantics for classic 0CFA will track literals (including lambdas) that appear in the program text. For integers that do not appear in the program text, the analysis uses a single token INT which conservatively approximates any possible integer. We note that many more abstractions for integers are possible, like integer ranges or symbolic binary relations among variables. These decisions do not interfere with AAM or abstract control, so we pick the simplest abstraction to simplify presentation.

The state space $\hat{\Sigma}$ for 0CFA is defined as:

$$
\begin{aligned}
v &: \widehat{Val} ::= l \mid f \mid INT \quad \text{(where } l \text{ and } f \text{ are drawn from the program text)} \\
\theta &: \widehat{Store} ::= Var \to \mathcal{P}\ (\widehat{Val}) \\
\hat{\sigma} &: \hat{\Sigma} ::= \mathcal{P}\ (Exp \times \widehat{Store})
\end{aligned}
$$

Because there are finite many literals in the program text, we can claim that $\widehat{Val}$ and $\widehat{Store}$ are finite.

The abstract semantics for $\hat{\Sigma}$ comes in two parts. First we define a *denotation function* $\mathscr{A}$ for *Atom* expressions.

$$
\begin{aligned}
\mathscr{A} &: \widehat{Store} \times Atom \to \mathscr{P}(\widehat{Val}) \\
\mathscr{A}(\theta, x) &\coloneqq \theta(x) \\
\mathscr{A}(\theta, l) &\coloneqq \{l\} \\
\mathscr{A}(\theta, \underline{\text{add1}}(a)) &\coloneqq \{\text{INT} \mid \exists i \sqsubseteq \text{INT} \in \mathscr{A}(\theta, a)\} \\
\mathscr{A}(\theta, \underline{\text{sub1}}(a)) &\coloneqq \{\text{INT} \mid \exists i \sqsubseteq \text{INT} \in \mathscr{A}(\theta, a)\} \\
\mathscr{A}(\theta, \underline{\text{gez}}(a)) &\coloneqq \{\text{TRUE} \mid \exists i \in \mathscr{A}(\theta, a) \wedge i >= 0\} \\
&\quad \cup \{\text{FALSE} \mid \exists i \in \mathscr{A}(\theta, a) \wedge i < 0\} \\
&\quad \cup \{\text{TRUE}, \text{FALSE} \mid \text{INT} \in \mathscr{A}(\theta, a)\} \\
\mathscr{A}(\theta, \underline{\lambda}(x) \to e) &\coloneqq \{\underline{\lambda}(x) \to e\} \\
\mathscr{A}(\theta, \underline{\lambda}(x)(k) \to e) &\coloneqq \{\underline{\lambda}(x)(k) \to e\}
\end{aligned}
$$

Second we define a *step relation* (as a function) $\mathscr{E}$ for *Exp* expressions.

$$
\begin{aligned}
\mathscr{E} &: Exp \times \widehat{Store} \to \mathscr{P}(Exp \times \widehat{Store}) \\
\mathscr{E}(\underline{\text{if}}(a)\{e_1\}\{e_2\}, \theta) &\coloneqq \{(e, \theta) \\
&\quad \mid e \in \{e_1 \mid \text{TRUE} \in \mathscr{A}(\theta, a)\} \cup \{e_2 \mid \text{FALSE} \in \mathscr{A}(\theta, a)\} \\
&\quad \} \\
\mathscr{E}(a_1(a_2, a_3), \theta) &\coloneqq \{(e, \theta') \\
&\quad \mid (\underline{\lambda}(x)(k) \to e) \in \mathscr{A}(\theta, a_1) \\
&\quad \mid v_2 \in \mathscr{A}(\theta, a_2) \\
&\quad \mid v_3 \in \mathscr{A}(\theta, a_3) \\
&\quad \mid \theta' \coloneqq \theta \sqcup [x \mapsto v_2] \sqcup [k \mapsto v_3] \\
&\quad \} \\
\mathscr{E}(\underline{\text{halt}}(a), \theta) &\coloneqq \{(\underline{\text{halt}}(a), \theta)\}
\end{aligned}
$$

(We omit the $(a_1(a_2))$ case. It is directly analogous to $(a_1(a_2, a_3))$).

The complete analysis of an expression e is defined as the least fixed point of a *collection semantics* for the relation $\mathscr{E}$:

$$\text{analysis} \coloneqq \mu(\hat{\sigma}) \to \{(e, \bot)\} \sqcup \mathscr{E}^\star(\hat{\sigma})$$

where

$$\mathscr{E}^\star : \mathscr{P}\,(Exp \times \widehat{Store}) \to \mathscr{P}\,(Exp \times \widehat{Store})$$
$$\mathscr{E}^\star(\hat{\sigma}) \coloneqq \bigcup_{e,\theta\in} \mathscr{E}(e,\theta)$$

A collecting semantics is used to track all states that the program could be in rather than just the final states.

3.2 Monadic 0CFA

The first insight in monadic abstract interpretation is to abbreviate the definitions of $\mathscr{A}$ and $\mathscr{E}$ using monadic state. We use a nondeterminism monad underneath a state monad transformer to write the analysis in monadic style. Monad transformers are just fancy names for simple types, and we write simple types underneath definitions which use monad transformer to aid understanding.

$$\mathscr{M} : Set \to Set$$
$$\mathscr{M}(a) \coloneqq \mathscr{S}_T(\widehat{Store})(\mathscr{P}\,)(a)$$
$$\mathscr{M}(a) \coloneqq \widehat{Store} \to \mathscr{P}\,(a \times \widehat{Store})$$

The monadic conversion of $\mathscr{A}$ interacts with the store θ using get and put operations.

$$\mathscr{A}_m : Atom \to \mathscr{M}(\mathscr{P}\,(\widehat{Val}))$$
$$\mathscr{A}_m(x) \coloneqq \text{do}$$
$$\theta \leftarrow \text{get-}\theta$$
$$\text{return}(\theta(x))$$
$$\mathscr{A}_m(l) \coloneqq \text{return}(\{l\})$$
$$\mathscr{A}_m(add1(a)) \coloneqq \text{do}$$
$$v \leftarrow \mathscr{A}_m(a)$$
$$\text{return}(\{\text{INT} \mid \exists i \sqsubseteq \text{INT} \in v\})$$
$$\mathscr{A}_m(sub1(a)) \coloneqq \text{do}$$
$$v \leftarrow \mathscr{A}_m(a)$$
$$\text{return}(\{\text{INT} \mid \exists i \sqsubseteq \text{INT} \in v\})$$
$$\mathscr{A}_m(gez(a)) \coloneqq \text{do}$$
$$v \leftarrow \mathscr{A}_m(a)$$
$$return(\{\text{TRUE} \mid \exists i \in v \wedge i >= 0\} \cup \{\text{FALSE} \mid \exists i \in v \wedge i < 0\} \cup \{\text{TRUE}, \text{FALSE} \mid INT \in v\})$$
$$\mathscr{A}_m(\underline{\lambda}(x)(k) \to e) \coloneqq \text{return}(\{\underline{\lambda}(x)(k) \to e\})$$

The monadic conversion of $\mathscr{E}$ uses $\uparrow \mathscr{P}$ to convert from powerset to monadic values. $\downarrow \mathbb{B}$ and $\downarrow \lambda_2$ are used to coerce lambdas out of abstract values, failing (with $\bot$) otherwise. The "effects" in $\mathscr{E}_m$ happen in calls to $\mathscr{A}_m$, $\uparrow \mathscr{P}$, $\downarrow \mathbb{B}$ and $\downarrow \lambda_2$.

$$
\begin{aligned}
&\mathscr{E}_m : Exp \to \mathscr{M}(Exp) \\
&\mathscr{E}_m(\underline{\text{if}}(a)\{e_1\}\{e_2\}) := \text{do} \\
&\qquad vP \leftarrow \mathscr{A}_m(a) \\
&\qquad v \leftarrow \uparrow \mathscr{P}(vP) \\
&\qquad b \leftarrow \downarrow \mathbb{B}(v) \\
&\qquad \text{return}(\{\text{if } b \text{ then } e_1 \text{ else } e_2\}) \\
&\mathscr{E}_m(a_1(a_2, a_3)) := \text{do} \\
&\qquad vP \leftarrow \mathscr{A}_m(a_1) \\
&\qquad v \leftarrow \uparrow \mathscr{P}(vP) \\
&\qquad (\underline{\lambda}(x)(k) \to e) \leftarrow \downarrow \lambda_2(v) \\
&\qquad vP_2 \leftarrow \mathscr{A}_m(a_2) \\
&\qquad vP_3 \leftarrow \mathscr{A}_m(a_3) \\
&\qquad \theta \leftarrow \text{get-}\theta \\
&\qquad \text{put-}\theta(\theta \sqcup [x \mapsto vP_2] \sqcup [k \mapsto vP_3]) \\
&\qquad \text{return}(e)
\end{aligned}
$$

$\uparrow \mathscr{P}$, $\downarrow \mathbb{B}$ and $\downarrow \lambda_2$ use the nondeterminism monadic effect and are defined as follows:

$$
\begin{aligned}
&\uparrow \mathscr{P} : \mathscr{P}(a) \to \mathscr{M}(a) \\
&\uparrow \mathscr{P}(xs) := \bigcup_{x \in xs} return(x)
\end{aligned}
$$

$$
\begin{aligned}
&\downarrow \mathbb{B} : \widehat{Val} \to \mathscr{M}(\mathbb{B}) \\
&\downarrow \mathbb{B}(b) := return(b) \\
&\downarrow \mathbb{B}(\_) := \bot
\end{aligned}
$$

$$
\begin{aligned}
&\downarrow \lambda_2 : \widehat{Val} \to \mathscr{M}(Lam) \\
&\downarrow \lambda_2(f) := return(f) \\
&\downarrow \lambda_2(\_) := \bot
\end{aligned}
$$

The monadic abstraction leads to a more compositional definition of the analysis. Pieces of the analysis which only use one effect need not mention the others. This compositionality will become more apparent when we extend this analysis to include context sensitivity.

As before, we must complete the analysis by building an abstract machine transition function:

$$\begin{aligned}\mathscr{E}_m^\star &: \mathscr{P}\,(Exp \times \widehat{Store}) \to \mathscr{P}\,(Exp \times \widehat{Store}) \\ \mathscr{E}_m^\star(\hat{\sigma}) &\coloneqq \bigcup_{e,\theta \in \hat{\sigma}} \mathscr{E}_m(e)(\theta)\end{aligned}$$

This definition of $\mathscr{M}$ and $\mathscr{E}_m^\star$ recovers exactly the analysis we wrote before.

3.3 Monadic kCFA

To incorporate context sensitivity (for $k = 1$) into the analysis we do the following:

- Add another state space component to $\hat{\Sigma}$ for environments: $\widehat{Env}$. This environment will track which calling function is responsible for binding a given parameter.
- Capture this environment when evaluating λ expressions, just the concrete semantics does for concrete environments.
- Add another state space component which tracks the program points visited so far by the program: $\widehat{Time}$.
- When interpreting a command $\mathscr{E}_m(e)$ set the time to e.
- When calling a function, record the current time next to the function arguments in the environment.

These steps have the effect of re-analyzing a function twice if it is called from two separate places.

Changing the state space two have two new components is achieved simply by adding two more state effects two our monad, one for $\widehat{Env}$ and one for $\widehat{Time}$. The resulting abstract state space looks like this:

$$\begin{aligned}\rho &: \widehat{Env} \coloneqq Var \rightharpoonup \widehat{Time} \\ \theta &: \widehat{Store} \coloneqq (Var, \widehat{Time}) \to \mathscr{P}\,(\widehat{Val}) \\ \tau &: \widehat{Time} \coloneqq Exp \text{ (the last program location visited)} \\ f, g &: Lam ::= \underline{\lambda}(x) \to e \mid \underline{\lambda}(x, k) \to e \\ clo &: Clo ::= \langle f, \rho \rangle \\ v &: \widehat{Val} ::= l \mid clo \mid INT \\ \hat{\sigma} &: \hat{\Sigma} \coloneqq \mathscr{P}\,(Exp \times \widehat{Time} \times \widehat{Env} \times \widehat{Store})\end{aligned}$$

Using monad transformers, we can easily add two new state components to our monadic interpreter.

$$\begin{aligned}
&\mathscr{M} \;:\; Set \to Set \\
&\mathscr{M}(a) \coloneqq \mathcal{S}_T(\widehat{Env})(\mathcal{S}_T(\widehat{Time})(\mathcal{S}_T(\widehat{Store})(\mathscr{P}\ ))(a) \\
&\mathscr{M}(a) \coloneqq \widehat{Env} \times \widehat{Time} \times \widehat{Store} \to \mathscr{P}\ (a \times \widehat{Env} \times \widehat{Time} \times \widehat{Store})
\end{aligned}$$

Capturing the environment when evaluating λ expressions is a simple use of monadic state:

$$\begin{aligned}
&\mathscr{A}_m \;:\; Atom \to \mathscr{M}(\mathscr{P}\ (\widehat{Val})) \\
&\quad ... \\
&\mathscr{A}_m(f) \coloneqq \text{do} \\
&\qquad \rho \leftarrow \text{get-}\rho \\
&\qquad \text{return}(\{\langle f, \rho \rangle\}) \\
&\quad ...
\end{aligned}$$

Moving abstract time forward is a simple update of the $\widehat{Time}$ state to be the current expression:

$$\begin{aligned}
&\mathscr{E}'_m \;:\; Exp \to \mathscr{M}(Exp) \\
&\mathscr{E}'_m(e) \coloneqq \text{do} \\
&\qquad \text{put-}\tau(e) \\
&\qquad \mathscr{E}_m(e)
\end{aligned}$$

Calling a function now uses time as an address for storing values in $\widehat{Store}$. This is analogous to the allocation concrete semantics shown in section 2.2.

$$
\begin{aligned}
&\mathscr{E}_m : Exp \to \mathscr{M}(Exp) \\
&... \\
&\mathscr{E}_m(a_1(a_2, a_3)) := \text{do} \\
&\quad \upsilon P \leftarrow \mathscr{A}_m(a_1) \\
&\quad \upsilon \leftarrow \uparrow \mathscr{P}(\upsilon P) \\
&\quad (\langle \underline{\lambda}(x)(k) \to e, \rho \rangle) \leftarrow \downarrow \lambda_2(\upsilon) \\
&\quad \upsilon P_2 \leftarrow \mathscr{A}_m(a_2) \\
&\quad \upsilon P_3 \leftarrow \mathscr{A}_m(a_3) \\
&\quad \theta \leftarrow \text{get-}\theta \\
&\quad \tau \leftarrow \text{get-}\tau \\
&\quad \text{put-}\rho(\rho[x \mapsto \tau][k \mapsto \tau]) \\
&\quad \text{put-}\theta(\theta \sqcup [(x, \tau) \mapsto \upsilon P_2] \sqcup [(k, \tau) \mapsto \upsilon P_3]) \\
&\quad \text{return}(e) \\
&... \\
&\mathscr{A}_m : Atom \to \mathscr{M}(\mathscr{P}(\widehat{Val})) \\
&\quad ... \\
&\mathscr{A}_m(x) := \text{do} \\
&\quad \rho \leftarrow \text{get-}\rho \\
&\quad \theta \leftarrow \text{get-}\theta \\
&\quad \text{return}(\theta(x, \rho(x))) \\
&\quad ...
\end{aligned}
$$

Finally, lookup for variables must follow the indirection of addresses through $\widehat{Store}$.

That's all there is to it. Adding two new states to the state space and instrumenting the analysis to track time leaves the rest of the semantics undisturbed. Without the monad abstraction, the entire semantics must be redefined when extending the analysis to be context sensitive.

So far we have used monad transformers to build analyses in a compositional manner. In the next section, we will see how to exploit the monadic abstraction to expose a tuning knob for abstract control.

4 Abstract Control

The instantiation for $\mathscr{M}$ in our previous examples happens to give a flow-sensitive path-sensitive analysis. We claim that our monad transformers compose in either direction,

and one might ask what happens when one swaps the order of state and nondeterminism transformers. Consider such a reordering of $\mathscr{M}$ (using the simpler 0CFA semantics):

$$\begin{aligned}\mathscr{M} &: \mathit{Set} \to \mathit{Set}\\ \mathscr{M}(a) &\coloneqq \mathscr{P}_T\ (\mathcal{S}(\widehat{\mathit{Store}}))(a)\\ \mathscr{M}(a) &\coloneqq \widehat{\mathit{Store}} \to \mathscr{P}\ (a) \times \widehat{\mathit{Store}}\end{aligned}$$

The implementations of $\mathscr{A}$ and $\mathscr{E}$ can be left as is, as they did not depend on the specific monad used, only the monadic effect interface. However, the type of $\mathscr{M}$ has changed so we must redefine $\mathscr{E}^\star_m$, as the old one is now ill-typed. One possible definition for $\mathscr{E}^\star_m$ is the following:

$$\begin{aligned}\mathscr{E}^\star_m &: \mathscr{P}\ (\mathit{Exp}) \times \widehat{\mathit{Store}} \to \mathscr{P}\ (\mathit{Exp}) \times \widehat{\mathit{Store}}\\ \mathscr{E}^\star_m(eP, \theta) &\coloneqq (\{e' \mid e' \in \pi_1(\mathscr{E}_m(e))(\theta)\ \mid\ e \in e\mathscr{P}\ \}, \bigcup_{e \in eP} \pi_2(\mathscr{E}(e))(\theta))\end{aligned}$$

This definition gives a flow-insensitive path-insensitive analysis. Note that the resulting state space $\widehat{\Sigma} = \mathscr{P}\ (\mathit{Exp}) \times \widehat{\mathit{Store}}$.

Another implementation of $\mathscr{E}^\star_m$ is actually possible for a different state space $\widehat{\Sigma} = \mathscr{P}\ (\mathit{Exp} \times \widehat{\mathit{Store}})$. This is the same $\widehat{\Sigma}$ from our first monadic definition of 0CFA, although we are using a different monad.

$$\begin{aligned}\mathscr{E}^\star_m &: \mathscr{P}\ (\mathit{Exp} \times \widehat{\mathit{Store}}) \to \mathscr{P}\ (\mathit{Exp} \times \widehat{\mathit{Store}})\\ \mathscr{E}^\star_m(\widehat{\sigma}) &\coloneqq \{(e', \pi_2(\mathscr{E}_m(e))(\theta))\ \mid\ e' \in \pi_1(\mathscr{E}_m(e))(\theta)\ \mid\ (e, \theta) \in \widehat{\sigma}\}\end{aligned}$$

This definition gives a flow-sensitive path-insensitive analysis.

By altering the monad behind the abstraction, we are able to tune the flow and path sensitivity of the analysis. The two choices for $\mathscr{M}$ alter to path sensitivity of the analysis. For the choice of $\mathscr{M} = \mathscr{P}_T\ (\mathcal{S}(\widehat{\mathit{Store}}))$, the two choices for $\widehat{\Sigma}$ alter the flow sensitivity of the analysis.

Our definition of a nondeterminism monad transformer is new in this work, and we give its definition here. Our nondeterminism monad transformer requires the underlying monad $\mathscr{M}$ to be a join-semilattice functor: $\mathscr{M}(a)$ must be a join-semilattice if a is a join-semilattice. We define the type $\mathscr{P}_T\ (\mathscr{M})(a) \coloneqq \mathscr{M}(\mathscr{P}\ (a))$ and define monad operations:

$$\begin{aligned}\mathit{return} &\coloneqq \mathit{return}_{\mathscr{M}} \circ \mathit{singleton}\\ \mathit{extend}(k) &\coloneqq \mathit{extend}_{\mathscr{M}}(\mathit{joins} \circ \mathit{map}_{\mathscr{P}}\ (k))\end{aligned}$$

where

$$\mathit{joins}(xs) \coloneqq \bigcup_{x \in xs} x$$

The fact that $\mathscr{P}_T\ (\mathscr{M})(a)$ is a join-semilattice follows trivially from the functorality of $\mathscr{M}$. Proofs of monad laws and other correctness properties are given in section 10.

5 Intensional Optimizations

Up to this point we have demonstrated a compositional framework for building abstract interpreters, and a method for exposing abstract control as a tuning knob for these analysis. Now we show how to implement two intentional optimizations, abstract garbage collection and MCFA, in a completely general setting.

5.1 Abstract Garbage Collection

Abstract garbage collection[10] is an optimization technique in abstract interpretation where unreachable abstract addresses are pruned from the state space. This is analagous to "real" garbage collection, where unreachable pointers are reclaimed for space efficiency. However, in abstract semantics, live addresses are *re-used* as part of the sound and finite approximation of an infinite address space. When an in-use address must be re-used for allocation, a static analysis is forced to lose precision. Abstract garbage collection exploits the fact that unreachable addresses can be reclaimed before they are re-used, resulting in no loss of precision when re-using an address.

For a generic implementation of garbage collection we assume an arbitrary $\mathscr{M}$ that has *get*, *put* and nondeterminism effects. This allows us to change $\mathscr{M}$, and therefore the flow and path sensitivity of the garbage collection, without having to reimplement our optimization.

We implement abstract garbage collection for our example of kCFA (for $k = 1$) described in section **??**. Our implementation of garbage collection uses two helper functions $\acute{t}\,Exp : Exp \to \mathscr{M}(\mathscr{P}\ (\widehat{Addr}))$ and $\acute{t}\,\widehat{Addr} : \widehat{Store} \to \widehat{Addr} \to \mathscr{P}\ (\widehat{Addr})$. $\acute{t}\,Exp$ is a monadic action which computes the live addresses reachable from the current expression using $\acute{t}\,\widehat{Addr}$. $\acute{t}\,\widehat{Addr}$ computes the addresses reachable from a single

address.

$$
\begin{aligned}
\text{t}Clo &: \widehat{Store} \times Clo \to \mathcal{P}(\widehat{Addr}) \\
\text{t}Clo(\theta, \langle f, \rho \rangle) &\coloneqq \bigcup_{x \in freevars(f)} \text{t}\widehat{Addr}(\theta)(\rho(x)) \\
\text{t}Addr &: \widehat{Store} \to \widehat{Addr} \to \widehat{Addr} \\
\text{t}\widehat{Addr}(\theta)(\ell) &\coloneqq \bigcup_{clo \in \theta(\ell)} \text{t}Clo(clo) \\
currClo &: Exp \to \mathcal{M}(Clo) \\
currClo(e) &\coloneqq \text{do} \\
&\qquad \rho \leftarrow \text{get-}\rho \\
&\qquad return(\langle \lambda(\_) \to e, \rho \rangle) \\
\text{t}Exp &: Exp \to \mathcal{M}(\widehat{Addr}) \\
\text{t}Exp(e) &\coloneqq \text{do} \\
&\qquad clo \leftarrow currClo(e) \\
&\qquad \text{t}Clo(clo)
\end{aligned}
$$

freevars computes the free variables of an expression in the usual way:

$$
\begin{aligned}
freevars(\underline{\text{if}}(a)\{e_1\}\{e_2\}) &\coloneqq freevars(a) \cup freevars(e_1) \cup freevars(e_2) \\
freevars(a_1(a_2)) &\coloneqq freevars(a_1) \cup freevars(a_2) \\
freevars(a_1(a_2, a_3)) &\coloneqq freevars(a_1) \cup freevars(a_2) \cup freevars(a_3) \\
freevars(\underline{\text{halt}}(a)) &\coloneqq freevars(a) \\
freevars(\underline{\lambda}(x) \to e) &\coloneqq freevars(e) - \{x\} \\
freevars(\underline{\lambda}(x, k) \to e) &\coloneqq freevars(e) - \{x, k\} \\
freevars(l) &\coloneqq \{\}
\end{aligned}
$$

Using helper functions $\text{t}Exp$ and $\text{t}\widehat{Addr}$, we can define garbage collection using the following strategy:

1. Find the set of addresses reachable from the current expression.
2. Find the least-fixed point of the reachability relation for the heap starting from this set.
3. Retain only the mappings in the heap that have keys in this set.

$$
\begin{aligned}
&gc : Exp \to \mathscr{M}(1) \\
&gc(e) := \text{do} \\
&\quad \theta \leftarrow \text{get-}\theta \\
&\quad \ell_0 \leftarrow t\text{Exp}(e) \\
&\quad \text{let } \ell_\infty := \mu(\ell) \to \ell_0 \sqcup \ell \sqcup t\text{Var}(\ell) \\
&\quad \text{put-}\theta(\bigcup_{x \in \ell_\infty} [x \mapsto (\theta(x))])
\end{aligned}
$$

This is literally the implementation of *concrete* garbage collection, but with control properties abstracted behind a monad. When instantiated with path-sensitive, flow-sensitive or flow insensitive monads, we can recover the appropriate version of control for abstract garbage collection.

In AAM, the authors describe abstract garbage collection for a path-sensitive CESK* machine. Later in the AAM story, the path-sensitive CESK* machine is abstracted to use a single global store, resulting in a less precise but more tractable analysis. However, the single-store variant of the analysis requires one to repeat the "instrument then abstract" design process all over again for abstract garbage collection. By abstracting control and exposing it as an orthogonal tuning knob, we are able to design flow and path sensitive variants of garbage collection with a single definition.

5.2 MCFA

MCFA[11] is an optimization that improves the asymptotic complexity of context-sensitive control flow analyses for functional languages. Context sensitive analysis were shown to be exponential for functional analyses, yet polynomial for OO analysis, creating an apparent paradox. MCFA resolves this paradox by identifying the difference in context sensitive analyses and porting the polynomial behavior from OO to functional analyses. The key insight of MCFA is to use packed, copied closures rather than linked closures in the abstract semantics. Like garbage collection, using packed rather than linked closures is a simple optimization a compiler or concrete interpreter might use to increase performance. Like *abstract* garbage collection, transporting this simple idea from concrete to abstract interpreters reveals a surprising analysis optimization.

Like abstract garbage collection, our implementation of MCFA also uses the semantics for kCFA. MCFA is implemented by creating a new packed environment, rather than just linking with the current environment: This reuses the definition of *freevars*

from the previous section.

$$
\begin{aligned}
&\mathscr{A}_m : Atom \to \mathscr{M}(\mathscr{P}(\widehat{Val})) \\
&\ldots \\
&\mathscr{A}_m(f) \coloneqq \text{do} \\
&\quad \text{let } ys \coloneqq \text{free-vars}(f) \\
&\quad vs \leftarrow map^\star(lookup)(ys) \\
&\quad \rho \leftarrow \{y \mapsto^\star v \mid (y, v) \in zip(ys, vs)\} \\
&\quad \text{return}(\langle f, \rho \rangle) \\
&\ldots
\end{aligned}
$$

Like abstract garbage collection, our implementation of MCFA is tunable from one implementation which has abstracted control through a monad. Flow and path sensitive variants of MCFA fall out of a simple rearranging of the monad transformer stack.

6 Correctness

One advantage of using our framework is that proofs of correctness for analyses are constructed compositionally and automatically. The proofs are compositional because they are established at the level of monad transformers, not hard-coded for a full monad stack. The proofs are derived automatically through the Galois-functorality of the monad transformers we use, which means they transport Galois connections in addition to monadic actions. Given the proofs provided by our framework, along with an argument about monotonicity of the monadic semantics, we can establish a proof of Galois connection for any given monadic instantiation.

To relate back to state machine semantics, we establish Galois connections between monadic actions in $\mathscr{M}$ and transitions functions for *some* abstract state space $\widehat{\Sigma}$, notated $(A \xrightarrow{mon} \mathscr{M}(B)) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\widehat{\Sigma}_T(A) \xrightarrow{mon} \widehat{\Sigma}_T(B))$. The abstract state space $\widehat{\Sigma}$is constructed from the monad transformer stack, although some transformer stacks support multiple abstract state spaces. Proofs of these properties are given in section 10.

Key lemmas in our construction of these Galois connections include:

- Proving that our new monad transformer for nondeterminism is a proper monad transformer. The key insight here is to require the underlying monad to be a join-semilattice functor, which holds for the analyses we discuss in this paper.
- Proving that both state and nondeterminism transformers transport Galois connections to abstract state machines. The key insight here is to move the Galois connection framework to a *functorial* framework, where $\alpha(1) = 1$ and $\alpha(g \circ f) = \alpha(g) \circ \alpha(f)$, and likewise for γ.

In our framework we also prove that $(\mathscr{S}_T(\jmath) \circ \mathscr{P}_T) \sqsubseteq (\mathscr{P}_T \circ \mathscr{S}_T(\jmath))$. These orders for the monad transformer stack correspond to path-sensitivity and path-insensitivity respectively.

For the $\mathscr{P}_T \circ \mathscr{S}_T(\mathscr{Z})$ ordering, we show two Galois connections are possible: one to the state space $\mathscr{P}\ (\_ \times \mathscr{Z})$ and one to the state space $(\mathscr{P}\ (\_) \times \mathscr{Z})$. These choices for Galois connection correspond to flow-sensitivity and flow-insensitivity respectively.

Using monad transformers and their relationships to state machines, we have shown independent of a particular language or analysis that:

$$\text{flow-insensitive} \sqsubseteq \text{flow-sensitive path-insensitive} \sqsubseteq \text{flow-sensitive}$$

A given *concrete* semantics will typically use the flow-sensitive path-sensitive monad. Using this already established ordering we can construct a Galois connection between a concrete semantics and a flow-insensitive abstract semantics with no additional effort.

This table summarizes our results, showing both the monad $\mathscr{M}$ and state space $\widehat{\Sigma}$ which correspond to each analysis:

Sensitivity	Path Sensitive	Path Insensitive
Flow Sensitive	$\mathscr{M} := \mathscr{S}_T \circ \mathscr{P}_T \mid \widehat{\Sigma} := \mathscr{P}_T \circ \times_T$	$\mathscr{M} := \mathscr{P}_T \circ \mathscr{S}_T \mid \widehat{\Sigma} := \mathscr{P}_T \circ \times_T$
Flow Insensitive	DNE	$\mathscr{M} := \times_T \circ \mathscr{P}_T \mid \widehat{\Sigma} := \times_T \circ \mathscr{P}_T$

where $\times_T$ is the state machine functor for cartesian product ($\times$).

7 Related Work

Abstract Control Johnson and Van Horn recently explored the concept of abstract control in their recent paper Abstracting Abstract Control (AAC)[6]. In this work, the authors discuss how to recover a pushdown abstraction systematically in the setting of AAM. Our work is orthogonal, and potentially compatible with theirs: our framework explains compositional analyses, while theirs explains pushdown control abstraction. We hope for an eventual combination of the two techniques, which would expose pushdown abstraction as a feature-independent tuning knob.

As far as the question "what is abstract control?", we paint a similar picture as do Johnson and Van Horn in AAC. In both works, abstract control is an abstraction over the fixpoint iteration of the semantic step function which classic AAM didn't account for. Our work gives one possible definition for what abstract control is: abstrct control is the combination of control effects used by an interpreter, coupled with the way these effects are related to abstract machines.

Compositional Approaches Another compositional approach to abstract interpretation is *calculational abstract interpretation*[1] as described by Cousot and Cousot. Calculational AI shares a similar philosophy to our work: abstract interpreters should be built piecewise from compositional components. In calculational AI, analyses are derived by applying a sequence of Galois connections to the concrete semantics. This contrasts to the approach of designing an analysis first, and manually justifying the Galois connection after. However, after applying a sequence of Galois connections, the analysis designer must *calculate* the computable analysis from the one derived by Galois connection composition, which in general is not computable.

The downfall of calculational AI is in the effort required during the calculation step. For example, Midgaard and Jensen[9] describe a calculational derivation of 0CFA, the same analysis described in this work, and it is a difficult process.

While work in calculational AI has perhaps seen less success than AAM, we note that the two approaches are actually compatible, tackling different parts of the analysis design process. AAM is a methodology for splitting the design process to first instrument a concrete semantics, and then systematically abstract. Both of these steps must be justified with Galois connections, and one could use calculation for either step. Our work provides computations and proofs for the abstraction step, but one might happily justify the instrumentation step via calculation.

Tuning Knobs Tuning knobs for static analysis are great when they're general enough for a wide range of applications. For example, work by Kastrinis and Smaragdakis[7][14] shows how to expose both context-sensitivity and object-sensitivity as orthogonal tuning knobs in an OO analysis. These knobs have shown to be sufficient for interesting applications like malware detection for Android applications[5]. ([5] directly uses the analysis framework described in[7].)

Path Sensitivity Path sensitivity in particular was shown to be profitible and tractable by Das et al.[3]. In this work the authors verify the absence of open/close file handle bugs in the GNU C compiler. Path sensitivity was crucial achieve the precision needed for their analysis, and their paper describes how to achieve this precision without too much added cost in complexity. These subtle variants of path sensitivity are important for applying path sensitive analysis in practice. We do not directly account for these particular variants of path sensitivity in our framework, although we hope to explore how they fit into our framework in future work.

8 Future Work

Fully Verified Compilation One direction of future work is to apply our approach to fully verified compilation. In fully verified compilation, the implementation of a comipler is typically embedded in proof assistant and proven correct all the way down to axioms in a logical system. This task is difficult due to both the extra rigor required in modeling and the proof engineering required in interacting with proof assistants. We hope the compositionality of the *correctness* of our approach will apply especially in this setting, because proofs need to be developed and maintained side-by-side with code.

CompCert[8] is perhaps the most well known fully verified compiler to date. CompCert compiles C programs down to machine code and performs simple local optimizations. A variation of CompCert which would support richer language features might benefit from our approach. Whole program analysis has also not been studied yet in the context of fully verified compilation, perhaps because establishing the proof of correctness is too burdensome. We hope our technique can be applied to a fully verified compiler for functional program which also performs advanced whole program analysis.

In the author's view, the biggest obstacle to implementing verified static analyses is the difficulty in establishing Galois connections using proof assistants. Galois connections sit in the middle-ground between computable functions (like an isomorphism) and mere specification (like a characteristic function). Arguments using Galois connections often start with some infinite set, after which calculation brings the result to something finite and computable. It is this process, moving from infinite to computable through calculation, that we conjecture has given experts the most difficulty in modeling Galois connections. All results to date of static analyses proven correct within proof assistants give up on establishing full Galois connection and settle for more direct methods. Tackling this problem could be a huge step forward in bringing more powerful analyses to the stage of fully verified compilation.

New Tuning Knobs for Analysis Another direction for future work is to discover more tuning knobs for analyses which have been missed by frameworks like AAM. For example, it would be nice if we could account for the pushdown abstraction from Johnson and Van Horn's work on Abstracting Abstract Control into our monadic formulation.

Less related to abstract control, we believe there is work to be done in porting object sensitivity to functional language analysis. We have preliminary results suggesting that object sensitivity can be seen as lexically scoped time in the functional setting. Exploring this would uncover yet another tuning knob to analysis designers to quickly interate through the analysis design process.

New Semantic Settings Our framework shares the small-step operational semantics roots of AAM. The small-step setting is convenient, but understood consequences when used as the basis for abstract interpretation which are less well understood.

For example, we can imagine a more denotational variant of this work where interpreters are related to a denotational space rather than to state machines. We have preliminary results suggesting that this can be done by allowing the monadic interpreter to be recursive, and then abstracting the meaning of recursion directly. The meaning of a program would then be given not given by an external fixpoint of the step function, but directly through a denotation for recursion. To recover an analysis, one would abstract a precise fixpoint finder like the Y-combinator with an approximating, finite fixpoint finder.

9 Conclusion

We have shown how to build compositional abstract interpreters which expose a tuning knob for abstract control. We use monad transformers as the building blocks for both constructing abstract interpreters and proving them correct piecewise. Using the monadic abstraction, we showed how to expose abstract control as a tuning knob to recover flow and path sensitivities for a static analysis. After exposing a tuning knob for abstract control, we then demonstrated how to implement two analysis optimizations which adapt effortlessly to changes in abstract control. Our approach is justified piecewise by relating monad transformers to state machine functors. To achieve this, we

develop a new monad transformer for nondeterminism and develop a functorial variant of Galois connection.

When using our framework, the analysis designer need only prove prove:

- The monadic interpreter $\mathscr{E}_m$ is monotonic.
- The semantics as written, including intensional optimizations, are correct.

After supplying these proofs, the analysis designer enjoys:

- An automatically derived analysis for their language with abstract control left as a tuning knob.
- A proof-by-construction of Galois connection (soundness and tightness) for the derived analysis.
- Easy integration of new analysis features, a la. AAM.
- Single-definition implementations of intensional optimizations like abstract garbage collection and MCFA.

10 Proofs

10.1 Definitions

Definition. For types $(\Sigma, \widehat{\Sigma} : \mathit{Set})$ which both have partial orders, a *Galois connection* between Σ and $\widehat{\Sigma}$, written $\Sigma \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \widehat{\Sigma}$, contains two operators:

$$\alpha : \Sigma \to \widehat{\Sigma}$$
$$\gamma : \widehat{\Sigma} \to \Sigma$$

which respect the following properties:

$$\alpha \text{is monotonic}$$
$$\gamma \text{is monotonic}$$
$$\text{contractive} : \alpha \circ \gamma \sqsubseteq id$$
$$\text{expansive} : id \sqsubseteq \gamma \circ \alpha$$

Definition. For a type $(F : \mathit{Set} \to \mathit{Set})$, we call F a *functor*, written $\mathit{Functor}(F)$, if one can define the operator:

$$\text{map} : \forall(A, B : \mathit{Set}), (A \to B) \to (F(A) \to F(B))$$

which respects the following properties:

$$\mathit{unit} : \mathit{map}(id) = id$$
$$\mathit{distributivity} : \mathit{map}(g \circ f) = \mathit{map}(g) \circ \mathit{map}(f)$$

Definition. For a type $(\mathscr{M} : Set \to Set)$, we call $\mathscr{M}$ a *monad*, written $Monad(\mathscr{M})$, if one can define two operators:

$$\begin{aligned} \text{return} &: \forall(A : Set), A \to \mathscr{M}(A) \\ \text{extend} &: \forall(A, B : Set), (A \to \mathscr{M}(B)) \to (\mathscr{M}(A) \to (\mathscr{M}(B))) \end{aligned}$$

which respect the following properties:

$$\begin{aligned} \text{left-unit} &: extend(return) = id \\ \text{right-unit} &: extend(k) \circ return = k \\ \text{associativity} &: extend(k_2) \circ extend(k_1) = extend(extend(k_2) \circ k_1) \end{aligned}$$

Corrolary 1. *All monads are functors.*

Definition. For a type $(\mathscr{T} : (Set \to Set) \to (Set \to Set))$, we call $\mathscr{T}$ a *monad transformer*, written $Transformer(\mathscr{T})$, if one can define a single operator:

$$lift : \forall(\mathscr{M} : Set \to Set)(A : Set), \mathscr{M}(A) \to \mathscr{T}(\mathscr{M})(A)$$

and the following property holds:

$$\forall(\mathscr{M} : Set \to Set), Monad(\mathscr{M}) \Rightarrow Monad(\mathscr{T}(\mathscr{M}))$$

Definition. For types $(\mathscr{s} : Set)$ and $(\mathscr{M} : Set \to Set)$, we call $\mathscr{M}$ a *monad state over* $\mathscr{s}$, written $MonadState(\mathscr{s})(\mathscr{M})$, if one can define operators:

$$\begin{aligned} \text{get} &: \mathscr{M}(\mathscr{s}) \\ \text{put} &: \mathscr{s} \to \mathscr{M}(1) \end{aligned}$$

Definition. For a type $(\mathscr{M} : Set \to Set)$, we call $\mathscr{M}$ a *monad plus*, written $MonadPlus(\mathscr{M})$, if the following property holds:

$$\forall(A : Set), JoinSemilattice(\mathscr{M}(A))$$

and the following additional properties hold:

$$\begin{aligned} \text{left-zero} &: extend(k)(\bot) = \bot \\ \text{right-zero} &: extend(const(\bot))(x) = \bot \\ \text{distributivity} &: extend(k)(x \sqcup y) = extend(k)(x) \sqcup extend(k)(y) \end{aligned}$$

Definition. For types $(\mathscr{M}, \widehat{\Sigma}_T : Set \to Set)$, we call $\mathscr{M}$ a *small step monad with state space* $\widehat{\Sigma}_T$, written $MonadSmallStep(\widehat{\Sigma}_T)(\mathscr{M})$, if one can define:

$$\forall(A, B : Set), (A \to \mathscr{M}(B)) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\widehat{\Sigma}_T(A) \to \widehat{\Sigma}_T(B))$$

Definition. For a type ($F : Set \to Set$) and property ($P : Set \to Prop$), we call F *functorial in* P, written $Functorial(P)(F)$, if the following property holds:

$$\forall(A : Set), P(A) \Rightarrow P(F(A))$$

and all operations in P distribute through monadic operations in F.

Example. For a type ($F : Set \to Set$) to be *functorial in* $JoinSemilattice$, the following additional laws must hold:

$$return(x \sqcup y) = return(x) \sqcup return(y)$$

10.2 ID

Definition. The *identity monad*, written ID, is defined:

$$ID : Set \to Set$$
$$ID(A) \coloneqq A$$

Lemma. *The identity monad is a monad with operators:*

$$return \coloneqq id$$
$$extend \coloneqq id$$

Proof. Unit and associativity laws are established trivially by definition. □

Lemma. *The identity monad is* functorial in all structures.

Proof. Holds by definition of ID. □

10.3 $\mathscr{P}_T$

Definition. The *set monad transformer*, written $\mathscr{P}_T$, is defined:

$$\mathscr{P}_T\ (\mathscr{M})(A) \coloneqq \mathscr{M}(\mathscr{P}\ (A))$$

Lemma. *For a given* ($\mathscr{M} : Set \to Set$) *where:*

- $\mathscr{M}$ *is a monad.*
- $\mathscr{M}$ *is functorial in JoinSemilattice.*

then $\mathscr{P}_T\ (\mathscr{M})$ *is a monad plus.*

Proof. By applying functoriality of $\mathscr{M}$ to the semilattice $\mathscr{P}$. □

Lemma. *For a given* ($\mathscr{M} : Set \to Set$) *where:*

- $\mathscr{M}$ *is a monad.*
- $\mathscr{M}$ *is functorial in JoinSemilattice.*

then $\mathscr{P}_T\ (\mathscr{M})$ *is a monad with operators:*

$$\begin{aligned} return &:= return_{\mathscr{M}} \circ singleton \\ extend(k) &:= extend_{\mathscr{M}}(joins \circ map_{\mathscr{P}}\ (k)) \end{aligned}$$

Proof. of left unit.

$$\text{left-unit} : extend(return) = id$$

$$\begin{aligned} extend(return) &= extend_{\mathscr{M}}(joins \circ map_{\mathscr{P}}\ (return_{\mathscr{M}} \circ singleton_{\mathscr{P}}\ )) \\ &\qquad \text{(definition of extend and join)} \\ &= extend_{\mathscr{M}}(\lambda(xs) \to \bigcup_{x \in xs} return_{\mathscr{M}}(singleton_{\mathscr{P}}\ (x))) \\ &\qquad \text{(definition of joins)} \\ &= extend_{\mathscr{M}}(\lambda(xs) \to return_{\mathscr{M}}(\bigcup_{x \in xs} singleton_{\mathscr{P}}\ (x))) \\ &\qquad \text{(functorality of semilattice in } \mathscr{M}) \\ &= extend_{\mathscr{M}}(\lambda(xs) \to return_{\mathscr{M}}(xs)) \qquad \text{(join identity for } \mathscr{P}\ ) \\ &= extend_{\mathscr{M}}(return_{\mathscr{M}}) \qquad (\eta \text{ reduction for } \lambda) \\ &= id \qquad \text{(left unit monad law for } \mathscr{M}) \end{aligned}$$

□

Proof. of right unit.

$$\text{right-unit} : extend(k)(return(x)) = k(x)$$

$$\begin{aligned} extend(k)(return(x)) &= extend_{\mathscr{M}}(joins \circ map_{\mathscr{P}}\ (k))(return_{\mathscr{M}}(singleton_{\mathscr{P}}\ (x))) \\ &\qquad \text{(definition of extend and join)} \\ &= joins(map_{\mathscr{P}}\ (k)(return_{\mathscr{M}}(singleton_{\mathscr{P}}\ (x)))) \\ &\qquad \text{(right unit monad law for } \mathscr{M}) \\ &= joins(return_{\mathscr{M}}(singleton_{\mathscr{P}}\ (k(x)))) \\ &\qquad \text{(map distribution law for } \mathscr{M} \text{ and } \mathscr{P}\ ) \\ &= k(x) \qquad \text{(definition of joins)} \end{aligned}$$

□

Proof. of associativity.

$$\text{associativity} : extend(k_2) \circ extend(k_1) = extend(extend(k_2) \circ k_1)$$

$$\begin{aligned}
extend(k_2) \circ extend(k_1) &= extend_{\mathscr{M}}(joins \circ map_{\mathscr{P}}\ (k_2)) \circ extend_{\mathscr{M}}(joins \circ map_{\mathscr{P}}\ (k_1)) \\
&\qquad \text{(definition of } extend\text{)} \\
&= extend_{\mathscr{M}}(extend_{\mathscr{M}}(joins \circ map_{\mathscr{P}}\ (k_2)) \circ joins \circ map_{\mathscr{P}}\ (k_1)) \\
&\qquad \text{(associativity law for } \mathscr{M}\text{)} \\
&= extend_{\mathscr{M}}(extend(k_2) \circ joins \circ map_{\mathscr{P}}\ (k_1)) \\
&\qquad \text{(definition of } extend\text{)} \\
&= extend_{\mathscr{M}}(\lambda(xs) \to extend(k_2)(\bigcup_{x \in xs} k_1(x))) \\
&\qquad \text{(definitions of } joins \text{ and } map_{\mathscr{P}}\text{)} \\
&= extend_{\mathscr{M}}(\lambda(xs) \to \bigcup_{x \in xs} extend(k_2)(k_1(x))) \\
&\qquad \text{(distributivity of } \cup \text{ for } \mathscr{M}\text{)} \\
&= extend_{\mathscr{M}}(joins \circ map_{\mathscr{P}}\ (extend(k_2) \circ k_1)) \\
&\qquad \text{(definition of } map_{\mathscr{P}}\text{)} \\
&= extend(extend(k_2) \circ k_1) \qquad \text{(definition of } extend\text{)}
\end{aligned}$$

□

Lemma. *For a given* ($\mathscr{M} : Set \to Set$) *and* ($\mathfrak{s} : Set$) *where:*

- $\mathscr{M}$ *is a monad.*
- $\mathscr{M}$ *is a monad state over* $\mathfrak{s}$*.*
- $\mathscr{M}$ *is functorial in JoinSemilattice.*

then $\mathscr{P}_T\ (\mathscr{M})$ *is a monad state over* $\mathfrak{s}$ *with operators:*

$$\begin{aligned}
get &:= map_{\mathscr{M}}(singleton)(get_{\mathscr{M}}) \\
put(\mathfrak{s}) &:= map_{\mathscr{M}}(singleton)(put_{\mathscr{M}}(\mathfrak{s}))
\end{aligned}$$

Lemma. *For a given* ($\mathscr{M} : Set \to Set$) *and* ($\widehat{\Sigma}_T : Set \to Set$) *where:*

- $\mathscr{M}$ *is a monad.*
- $\mathscr{M}$ *is a small-step monad over* $\widehat{\Sigma}_T$*.*
- $\mathscr{M}$ *is functorial in join semilattice.*
- $\widehat{\Sigma}_T$ *is a functor.*

then $\mathscr{P}_T\ (\mathscr{M})$ *is a small step monad over* $(\widehat{\Sigma}_T \circ \mathscr{P}\ )$ *with Galois connection maps:*

$$\alpha\ :\ \forall(A, B\ :\ Set), (A \to \mathscr{P}_T\ (\mathscr{M})(B)) \to (\widehat{\Sigma}_T(\mathscr{P}\ (A)) \to \widehat{\Sigma}_T(\mathscr{P}\ (B)))$$
$$\alpha(f) \coloneqq \alpha_{\mathscr{M}}(joins \circ map_{\mathscr{P}}\ (f))$$

$$\gamma\ :\ \forall(A, B\ :\ Set), (\widehat{\Sigma}_T(\mathscr{P}\ (A)) \to \widehat{\Sigma}_T(\mathscr{P}\ (B))) \to (A \to \mathscr{P}_T\ (\mathscr{M})(B))$$
$$\gamma(f) \coloneqq \gamma_{\mathscr{M}}(f \circ map_{\widehat{\Sigma}_T}(return_{\mathscr{P}}\ ))$$

$$\alpha \circ \gamma \sqsubseteq id$$

$$(\alpha \circ \gamma)(f) = \alpha_{\mathscr{M}}(joins \circ map_{\mathscr{P}}\ (\gamma_{\mathscr{M}}(f \circ map_{SS}(return_{\mathscr{P}}\ ))))$$

References

[1] Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[2] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238--252, New York, NY, USA, 1977. ACM.

[3] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, volume 37 of *PLDI '02*, pages 57--68, New York, NY, USA, May 2002. ACM.

[4] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235--271, 1992.

[5] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-Based detection of android malware through static analysis. In *SIGSOFT FSE*, 2014.

[6] J. Ian Johnson and David Van Horn. Abstracting abstract control (extended). August 2014.

[7] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 423--434, New York, NY, USA, 2013. ACM.

[8] Xavier Leroy. Formal verification of a realistic compiler. *Communication of the ACM*, 52(7):107--115, July 2009.

[9] Jan Midtgaard and Thomas Jensen. A calculational approach to Control-Flow analysis by abstract interpretation static analysis. In María Alpuente and Germán Vidal, editors, *Static Analysis*, volume 5079 of *Lecture Notes in Computer Science*, chapter 23, pages 347--362. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008.

[10] Matthew Might and Olin Shivers. Improving flow analyses via Gamma-CFA: Abstract garbage collection and counting. In *ICFP '06: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, pages 13--25, New York, NY, USA, 2006. ACM.

[11] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 305--315. ACM Press, 2010.

[12] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[13] Olin G. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.

[14] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 46 of *POPL '11*, pages 17--30, New York, NY, USA, January 2011. ACM.

[15] David Van Horn and Matthew Might. Abstracting abstract machines. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, volume 45 of *ICFP '10*, pages 51--62, New York, NY, USA, September 2010. ACM.