Abstract interpreters for free

Matthew Might

University of Utah, Salt Lake City, Utah, USA,
might@cs.utah.edu, http://matt.might.net/

Abstract. In small-step abstract interpretations, the concrete and abstract semantics bear an uncanny resemblance. In this work, we present an analysis-design methodology that both explains and exploits that resemblance. Specifically, we present a two-step method to convert a small-step concrete semantics into a family of sound, computable abstract interpretations. The first step re-factors the concrete state-space to eliminate recursive structure; this refactoring of the state-space simultaneously determines a store-passing-style transformation on the underlying concrete semantics. The second step uses inference rules to generate an abstract state-space and a Galois connection simultaneously. The Galois connection allows the calculation of the "optimal" abstract interpretation. The two-step process is unambiguous, but nondeterministic: at each step, analysis designers face choices. Some of these choices ultimately influence properties such as flow-, field- and context-sensitivity. Thus, under the method, we can give the emergence of these properties a graph-theoretic characterization. To illustrate the method, we systematically abstract the continuation-passing style lambda calculus to arrive at two distinct families of analyses. The first is the well-known k-CFA family of analyses. The second consists of novel "environment-centric" abstract interpretations, none of which appear in the literature on static analysis of higher-order programs.

1 Introduction: Can we get two for the price of one?

In small-step abstract interpretation [4, 5, 16], there is often a tight correspondence between the concrete and abstract semantics. When one implements a small-step interpreter and then a small-step static analyzer, the correspondence is so obvious that there is a "nagging sense" of duplicated effort—large tracts of code for the analyzer and the interpreter end up looking *almost* identical. Suffering this *déjà vu* long enough leads one to ask:

> Is there a principled method for constructing a sensible abstract interpretation of a small-step concrete semantics automatically?

As we will demonstrate, the answer is *yes*: for any given small-step concrete semantics, there exist "natural" abstract interpretations, and there is a procedure an analysis designer can execute to construct these analyses.

By applying our method to the concrete semantics for continuation-passing style, we end up discovering both known analyses (like k-CFA) and unknown

analyses (which take a fundamentally different approach to abstraction of environments and closures). Choice points in the method also end up (quite unexpectedly) providing graph-theoretic explanations for the emergence of properties such as flow-, field- and context-sensitivity (Section 6).

An additional benefit of the method is pedagogical: it adds a more formal dimension to the art of analysis design. One can teach a student *what* abstract interpretation is, and *what* Galois connections are, but this knowledge doesn't make a student an analysis designer any more than rote knowledge of the syntax of Java makes her a programmer. She is still left with the question of *how* to design a static analysis. The method described in this work provides one answer to that question: it constitutes a process students can follow to go from a concrete semantics to an abstract interpreter.

1.1 An example to illustrate correspondence and redundancy

A brief example informally illustrates the degree to which the abstract semantics resemble the concrete semantics. We point out this resemblance to encourage the idea that the abstract semantics might be synthesized from the concrete semantics. Consider the concrete rule for MOVE in a register machine:

$$(\llbracket var := var' \rrbracket : \boldsymbol{stmt}, env, heap) \Rightarrow (\boldsymbol{stmt}, env[var \mapsto env(var')], heap).$$

The transition moves to the next statement, and updates the environment in the process. Contrast this concrete rule with the "abstract" rule for MOVE:

$$(\llbracket var := var' \rrbracket : \boldsymbol{stmt}, \widehat{env}, \widehat{heap}) \leadsto (\boldsymbol{stmt}, \widehat{env}[var \mapsto \widehat{env}(var')], \widehat{heap}).$$

This rule is suspiciously similar to the concrete one. In fact, the rules are so similar that presenting them both in a technical paper begs charges of redundancy. Implementing them both in code looks like the "copy-and-paste" anti-pattern.

With other rules, the correspondence is less direct. Consider the concrete rule for pointer assignment:

$$(\llbracket *var := var' \rrbracket : \boldsymbol{stmt}, env, heap) \Rightarrow (\boldsymbol{stmt}, env, heap[env(var) \mapsto env(var')]),$$

and its abstract counterpart:

$$\frac{\hat{a} \in \widehat{env}(var)}{(\llbracket *var := var' \rrbracket : \boldsymbol{stmt}, \widehat{env}, \widehat{heap}) \leadsto (\boldsymbol{stmt}, \widehat{env}, \widehat{heap} \sqcup [\hat{a} \mapsto \widehat{env}(var')]).}$$

In contrast with the concrete rule, the abstract rule is nondeterministic—there is one subsequent state for each possible abstract address to which the machine may write. The abstract rule also changed from functional extension to join for updating the heap. Staring at the similarities, it *feels* like there should be a principled method that can figure out where to introduce the nondeterminism and where to swap functional extension for join.

1.2 The two-step method: Snipping and trickling

We will describe a process for converting a small-step concrete semantics into a parameterized abstract semantics. At high level, the process has two steps:

1. The first step **snips** recursive structure out of concrete state-space. While state-spaces with recursive structure *can* be abstracted, it's much easier to abstract state-spaces without recursive structure. To perform the "snip," we view the concrete state-space as a dependence graph. Snipping selectively cuts cycle-forming edges in this graph. Each cut induces a corresponding store-passing-style transformation [17] of the concrete semantics.

2. The second step **trickles** abstraction up the concrete state-space, starting with the leaves of the DAG left over from the snipping operation. The designer must choose a specific abstraction for these leaves. Then, to automate remainder of the process, we recursively apply inference rules that form Galois connections [5]. A Galois connection inference rule has the form, "If the structures X and Y form a Galois connection, the structure $F(X, Y)$ is also a Galois connection," for some functor F. Consequently, these inference rules "trickle up" abstraction from the leaves of the concrete state-space. Once the rules infer a top-level Galois connection between concrete and abstract states, we can calculate the "optimal" abstract interpretation.[1]

The rationale for these two steps comes from an observation on the design of abstract interpretations—finite abstract state-spaces are easier to work with, because no widening is necessary in order to achieve termination. Yet, in order for a small-step semantics to describe a Turing-complete system, the state-space for the small-step semantics must have infinite size. Thus, the motivation for the two-step process is to effect a systematic compaction from an infinite to a finite state-space.

The first step (snipping) exposes the source of the unboundedness of the concrete state-space; it then isolates this unboundedness to the leaf nodes in a dependence graph over the state-space. The second step (trickling) starts by abstracting these leaf nodes into finite sets. Because the snipped concrete state-space lacks recursion, if the abstractions on these leaves are finite, the resulting abstract state-space is also finite.

2 Continuation-passing-style λ-calculus

For the sake of grounding our discussion in specific examples, we'll look at the continuation-passing style λ-calculus (CPS). We will gradually transform the

[1] The word *optimal* has to be qualified: optimal under what constraints? With Galois connections [5], the calculated analysis is *optimal* with respect to the specific abstraction embodied by the Galois connection. Every Galois connection implies many sound analyses, but only one of these is the most precise, and this analysis can be calculated by composing the concretization function with the concrete semantics and again with the abstraction function. That is, the optimal analysis appears to concretize the input, run the exact semantics, and then abstract the output.

concrete semantics for CPS into several abstract interpreters. The grammar for (pure) CPS is conveniently small:

$$\begin{aligned} f, e \in \mathsf{Exp} &::= \mathsf{Var} + \mathsf{Lam} \\ \mathit{lam} \in \mathsf{Lam} &::= (\lambda\ (v_1 \ldots v_n)\ \mathit{call}) \\ v &\in \mathsf{Var} \text{ is a set of identifiers} \\ \mathit{call} \in \mathsf{Call} &::= (f\ e_1 \ldots e_n). \end{aligned}$$

A textbook concrete (small-step) state-space (Σ) for pure CPS is also simple:

$$\begin{aligned} \varsigma \in \Sigma &= \mathsf{Call} \times \mathit{Env} \\ \rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathit{Clo} \\ \mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{Env}. \end{aligned}$$

And, the small-step transition relation, $(\Rightarrow) \subseteq \Sigma \times \Sigma$ needs but one rule:

$$\begin{aligned} (\llbracket (f\ e_1 \ldots e_n) \rrbracket, \rho) \Rightarrow (\mathit{call}, \rho''), \text{ where} \\ (\mathit{lam}, \rho') &= \mathcal{E}(f, \rho) \\ \mathit{lam} &= \llbracket (\lambda\ (v_1 \ldots v_n)\ \mathit{call}) \rrbracket \\ \rho'' &= \rho'[v_i \mapsto \mathcal{E}(e_i, \rho)], \end{aligned}$$

where the argument evaluator $\mathcal{E} : \mathsf{Exp} \times \mathit{Env} \rightharpoonup \mathit{Clo}$ evaluates an expression in the context of an environment:

$$\begin{aligned} \mathcal{E}(v, \rho) &= \rho(v) \\ \mathcal{E}(\mathit{lam}, \rho) &= (\mathit{lam}, \rho). \end{aligned}$$

3 A naïve attempt: "Throw hats on everything"

At first glance, it appears that the only change between concrete and abstract semantics is typographical: hats appear on all of the abstract domains (and the ranges of some functions become, somewhat mysteriously, power domains). Inspired by this observation, we can try it with the domains for continuation-passing style, to arrive at an abstract state-space $\hat{\Sigma}$:

$$\begin{aligned} \hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Call} \times \widehat{\mathit{Env}} \\ \hat{\rho} \in \widehat{\mathit{Env}} &= \mathsf{Var} \to \mathcal{P}\left(\widehat{\mathit{Clo}}\right) \\ \widehat{\mathit{clo}} \in \widehat{\mathit{Clo}} &= \mathsf{Lam} \times \widehat{\mathit{Env}}. \end{aligned}$$

But, there is an obvious problem with this "abstract" state-space: it's infinite, because closures contain environments, and environments contain closures. Moreover, a structural abstraction function defined on the concrete state-space isn't

well-founded; there is always the possibility (in theory) that it will encounter an infinite closure, such as clo_∞:

$$clo_\infty = (lam, [v \mapsto \{clo_\infty\}]).$$

Abstract interpreters typically operate over finite state-spaces in order to guarantee termination. For infinite abstract state-spaces, widening can accelerate and guarantee convergence, but a widening operator has to be defined on a case-by-case basis. Constructing an appropriate widening operator is not a process that can be fully mechanized; it requires creativity and intuition. And, in this case, there is no obvious widening operator.

Instead of widening, we choose to eliminate recursion from the state-space through an automatable process called "snipping the knots." Once recursion is eliminated from the concrete state-space, we can systematically transform it into an abstract state-space, starting with its leaves and abstracting upward.

4 Step 1: Snipping the knots with store-passing style

Recursive structures pose problems with well-foundedness for mathematicians. Because they are difficult to abstract "directly," they also pose a problem for abstract interpretation. Yet, in computer programming, recursive structures—even infinitely recursive structures—are neither uncommon nor troublesome. Every first-year computer science student knows how to build recursive data structures: pointers.

If we view a semantics as an interpreter, then we can exploit this freshman insight to eliminate recursion from mathematical structures as well—we can introduce a store and pointers into a small-step semantics. Specifically, we can use an off-the-shelf store-passing style transformation of the concrete semantics [17], and then thread recursive structure through the store.

To prepare for store-passing style, we represent the concrete definition of the state-space as a graph with edges from uses to definitions of each set (Figure 1). For example, in CPS, we add edges from the node Σ to the node Call and to the node Env, because the definition of the set Σ[2] refers to both Call and Env; for the same reason, we add edges from the node Clo to the node Lam and to the node Env.[3] Once in dependence-graph form, we must choose a set of edges to "snip" in order to eliminate cycles from the graph.

To eliminate cycles in the concrete state-space for CPS (Figure 1), we can snip this graph in either of two places: we can snip the edge from the node Clo

[2] $\Sigma = \mathsf{Call} \times Env$.

[3] The observant reader might wonder why we omit dependence edges between syntax nodes, *e.g.*, from Lam to Call and *vice versa*. In fact, we could add them. However, we will only operate on programs of finite size, and on subterms of the original program. As a result, syntax never contributes to the unboundedness of the concrete state-space; hence, there is no reason to snip these edges. If we used a substitution-/reduction-based concrete semantics, which could introduce new terms during execution, then we would have to add and snip these edges as well.

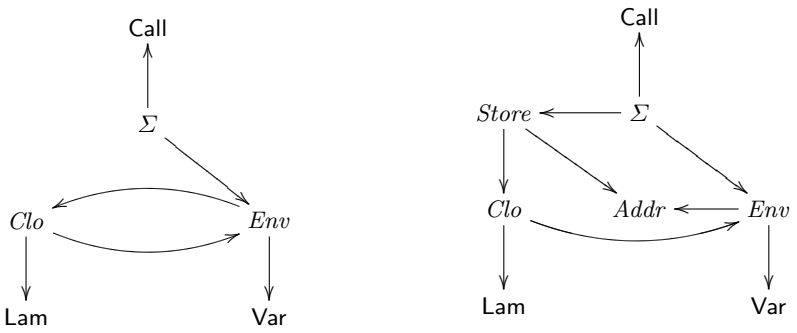


Fig. 1. A dependence graph of the concrete state-space for CPS before a snip (left) and after snipping the $Env \to Clo$ edge (right). After the snip, there are no longer cycles in the dependence graph.

to the node *Env*, or we can snip the edge from the node *Env* to the node *Clo*. It doesn't matter whether we snip one edge or both; the final result will still be a sound abstract interpretation. Snipping the $Env \to Clo$ edge will end up giving us k-CFA [18, 19]. Snipping the $Clo \to Env$ edge will end up giving us a novel and interesting hierarchy of control-flow analyses which, to the author's knowledge, has not appeared elsewhere.

4.1 Making a snip

To make a snip, we need to add a store to the concrete state-space, and then thread this store through the transition relation. To snip an edge going from a set A to a set B, we redirect the snipped edge from its original target to a newly created (infinite) set of addresses *Addr*. We then add the original target to the range of the store *Store*, so that:

$$\sigma \in Store = Addr \rightharpoonup B.$$

Before performing a standard store-passing style transformation on the semantics, the store is made a component of each state.

4.2 Option 1: Snipping $Env \to Clo$

Snipping the $Env \to Clo$ edge of the CPS semantics and applying the naïve, mechanical store-passing transform to the concrete semantics yields the state-space dependence graph in Figure 1 and the following state-space:

$$\begin{aligned} \varsigma \in \Sigma &= \mathsf{Call} \times Env \times Store \\ \rho \in Env &= \mathsf{Var} \rightharpoonup Addr \\ clo \in Clo &= \mathsf{Lam} \times Env \\ \sigma \in Store &= Addr \rightharpoonup Clo \\ a \in Addr &\text{ is an infinite set of addresses,} \end{aligned}$$

and a new transition rule:

$$\begin{aligned}(\llbracket(f\ e_1 \ldots e_n)\rrbracket, \rho, \sigma) &\Rightarrow (\mathit{call}, \rho'', \sigma''), \text{ where} \\ ((\mathit{lam}, \rho'), \sigma'_0) &= \mathcal{E}((f, \rho), \sigma) \\ \mathit{lam} &= \llbracket(\lambda\ (v_1 \ldots v_n)\ \mathit{call})\rrbracket \\ a_1, \ldots, a_n &\notin \mathit{dom}(\sigma'_0) \\ \rho'' &= \rho'[v_i \mapsto a_i] \\ (\mathit{clo}_i, \sigma'_i) &= \mathcal{E}((e_i, \rho), \sigma'_{i-1}) \\ \sigma'' &= \sigma'_n[a_i \mapsto \mathit{clo}_i],\end{aligned}$$

where the argument evaluator $\mathcal{E} : (\mathsf{Exp} \times \mathit{Env}) \times \mathit{Store} \rightharpoonup (\mathit{Clo} \times \mathit{Store})$ evaluates an expression in the context of an environment and a store, to return a value and a store:

$$\begin{aligned}\mathcal{E}((v, \rho), \sigma) &= (\sigma(\rho(v)), \sigma) \\ \mathcal{E}((\mathit{lam}, \rho), \sigma) &= ((\mathit{lam}, \rho), \sigma).\end{aligned}$$

Cleaning up with useless-variable elimination Applying useless-variable elimination [20] to the transformed semantics (again treating the semantics like an interpreter) picks up on the fact that the argument evaluator never modifies the store, which leads to a cleaner transition relation:

$$\begin{aligned}(\llbracket(f\ e_1 \ldots e_n)\rrbracket, \rho, \sigma) &\Rightarrow (\mathit{call}, \rho'', \sigma'), \text{ where} \\ (\mathit{lam}, \rho') &= \mathcal{E}(f, \rho, \sigma) \\ \mathit{lam} &= \llbracket(\lambda\ (v_1 \ldots v_n)\ \mathit{call})\rrbracket \\ a_1, \ldots, a_n &\notin \mathit{dom}(\sigma) \\ \rho'' &= \rho'[v_i \mapsto a_i] \\ \mathit{clo}_i &= \mathcal{E}(e_i, \rho, \sigma) \\ \sigma' &= \sigma[a_i \mapsto \mathit{clo}_i],\end{aligned}$$

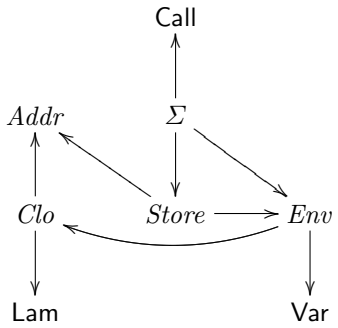
where the argument evaluator $\mathcal{E} : \mathsf{Exp} \times \mathit{Env} \times \mathit{Store} \rightharpoonup \mathit{Clo}$ evaluates an expression in the context of an environment and a store to return a value:

$$\begin{aligned}\mathcal{E}(v, \rho, \sigma) &= \sigma(\rho(v)) \\ \mathcal{E}(\mathit{lam}, \rho, \sigma) &= (\mathit{lam}, \rho).\end{aligned}$$

4.3 Option 2: Snipping *Clo* → *Env*

The other option for eliminating recursion is to snip the $\mathit{Clo} \rightarrow \mathit{Env}$ edge. This snip leads to a family of analyses with a character unlike any in the published literature on higher-order flow analysis.

Snipping this edge and performing the store-passing transform leads to the following state-space dependence diagram:



and the state-space:

$$\begin{aligned}
\varsigma \in \Sigma &= \mathsf{Call} \times \mathit{Env} \times \mathit{Store} \\
\rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathit{Clo} \\
\mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{Addr} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup \mathit{Env} \\
a \in \mathit{Addr} &\text{ is an infinite set of addresses,}
\end{aligned}$$

and the following transition rule:

$$\begin{aligned}
(\llbracket (f\ e_1 \ldots e_n) \rrbracket, \rho, \sigma) \Rightarrow (\mathit{call}, \rho'', \sigma'), \text{ where} \\
a \notin \mathit{dom}(\sigma) \\
\sigma' = \sigma[a \mapsto \rho] \\
(\mathit{lam}, a') = \mathcal{E}(f, a, \sigma') \\
\mathit{lam} = \llbracket (\lambda\ (v_1 \ldots v_n)\ \mathit{call}) \rrbracket \\
\mathit{clo}_i = \mathcal{E}(e_i, a, \sigma') \\
\rho'' = (\sigma(a'))[v_i \mapsto \mathit{clo}_i],
\end{aligned}$$

where the argument evaluator $\mathcal{E} : \mathsf{Exp} \times \mathit{Addr} \times \mathit{Store} \rightharpoonup \mathit{Clo}$ evaluates an expression in the context of an environment's address and a store to return a value:

$$\begin{aligned}
\mathcal{E}(v, a, \sigma) &= \sigma(a)(v) \\
\mathcal{E}(\mathit{lam}, a, \sigma) &= (\mathit{lam}, a).
\end{aligned}$$

4.4 Optional snips

Of course, one can also snip non-cycle-forming edges. Under the next stage in the method (trickle-up abstraction), these optional snips manifest themselves as knobs that tune some well-known properties such as field-sensitivity (if one snips

the $Env \rightarrow \mathsf{Var}$ edge) and flow-sensitivity (if one snips the $\Sigma \rightarrow \mathsf{Call}$ edge). Yet other snips (such as the $Clo \rightarrow \mathsf{Lam}$ edge) create knobs for tuning the precision and speed of the analysis which don't appear anywhere in the literature.

Finally, we point out that one can snip as many or as few edges in the dependence graph as desired, so long as the resulting dependence graph is acyclic.

5 Step 2: Trickling up abstraction

Once snips have eliminated recursive structure from the concrete state-space (Σ), we need (1) an abstract state-space ($\hat{\Sigma}$), and (2) a Galois connection between the concrete state-space and the abstract state-space ($\mathcal{P}(\Sigma) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{P}(\hat{\Sigma})$). Once we have the Galois connection, a foundational result by the Cousots [5] enables us to *calculate* an "optimal" small-step abstract transition relation: $(\rightsquigarrow) = \alpha \circ (\Rightarrow) \circ \gamma$.

5.1 Abstracting the leaves of the state-space dependence graph

To generate the abstract state-space, we focus initially on the leaves of the dependence graph for the concrete state-space. We require that the analysis designer choose a finite set $\hat{A}$ for each leaf node A; these finite sets will become the leaves of the abstract state-space. For each concrete leaf set A, the analysis designer must also specify an extraction function $\eta : A \rightarrow \hat{A}$ that maps a concrete element to an abstract element. Once the extraction function is fixed, we can automate the synthesis of the abstract state-space with inference rules that build structural Galois connections.

It is straightforward to convert an extraction function into a Galois connection [13]. Specifically, given a surjective map $\eta : A \rightarrow \hat{A}$, the structure $(\mathcal{P}(A), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{P}(\hat{A}), \subseteq)$, where:

$$\alpha(S) = \{\eta(a) : a \in S\}$$
$$\gamma(\hat{S}) = \left\{a : \hat{a} \in \hat{S} \text{ and } \eta(a) = \hat{a}\right\},$$

forms a Galois connection.

In practice, snipping and store-passing style transforms will leave an infinite leaf node in the form of the set of addresses. In this case, the extraction function on addresses fixes the polyvariance and the context-sensitivity of the analysis [9].

5.2 Recursively constructing the abstract state-space

To synthesize the abstract state-space automatically, we will utilize inference rules. These inference rules will build up structural Galois connections. In particular, these rules will take the Galois connections defined on leaves, and percolate them up to a top-level Galois connection over sets of states.

Most of the inference rules have the form "if structures $X_1, X_2, \ldots, X_n$ are Galois connections, then $F(X_1, X_2, \ldots, X_n)$ is also a Galois connection (for some functor F)."

Example 1. Given Galois connections $(A, \sqsubseteq_A) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\hat{A}, \sqsubseteq_{\hat{A}})$ and $(B, \sqsubseteq_B) \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} (\hat{B}, \sqsubseteq_{\hat{B}})$, the product Galois connection is the structure $(A \times B, \sqsubseteq_{A\times B}) \underset{\alpha''}{\overset{\gamma''}{\leftrightarrows}} (\hat{A} \times \hat{B}, \sqsubseteq_{\hat{A}\times\hat{B}})$, where:

$$\alpha''(a, b) = (\alpha(a), \alpha'(b))$$
$$\gamma''(\hat{a}, \hat{b}) = (\gamma(a), \gamma'(b)).$$

For the sake of mechanizing the process, we phrase the definitions of structural Galois connections as inference rules taking us from less-structured Galois connection to a more-structured one; for example:

$$\frac{(A, \sqsubseteq_A) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\hat{A}, \sqsubseteq_{\hat{A}}) \qquad (B, \sqsubseteq_B) \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} (\hat{B}, \sqsubseteq_{\hat{B}})}{(A \times B, \sqsubseteq_{A\times B}) \underset{\alpha''}{\overset{\gamma''}{\leftrightarrows}} (\hat{A} \times \hat{B}, \sqsubseteq_{\hat{A}\times\hat{B}}).}$$

5.3 Galois inference rules

In this work, we use the inference rules sketched in Figure 2 in addition to the "standard" structural Galois connections found in Nielson *et al.* [13]. (For brevity, we omit defining new concretization and abstraction maps in each rule.)

$$(\mathcal{P}(A), \sqsubseteq_1) \underset{\lambda S.S}{\overset{\lambda S.S}{\leftrightarrows}} (\mathcal{P}(A), \sqsubseteq_1) \qquad \text{(power identity)}$$

$$\frac{(\mathcal{P}(A), \sqsubseteq_1) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{P}(\hat{A}), \sqsubseteq_2) \qquad (\mathcal{P}(B), \sqsubseteq_1') \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} (\mathcal{P}(\hat{B}), \sqsubseteq_2')}{(\mathcal{P}(A \times B), \sqsubseteq_1'') \underset{\alpha''}{\overset{\gamma''}{\leftrightarrows}} (\mathcal{P}(\hat{A} \times \hat{B}), \sqsubseteq_2'')} \qquad \text{(power product)}$$

$$\frac{(\mathcal{P}(Y), \sqsubseteq_1) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{P}(\hat{Y}), \sqsubseteq_2)}{(\mathcal{P}(X \to Y), \sqsubseteq_1'') \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} (\mathcal{P}(X \to \hat{Y}), \sqsubseteq_2'')} \qquad \text{(image)}$$

$$\frac{(\mathcal{P}(X), \sqsubseteq_1) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\hat{X}, \sqsubseteq_2)}{(\mathcal{P}(X), \sqsubseteq_1) \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} (\mathcal{P}(\hat{X}), \sqsubseteq_2')} \qquad \text{(power lift)}$$

$$\frac{(\mathcal{P}(X), \sqsubseteq_1) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{P}(\hat{X}), \sqsubseteq_2) \qquad (\mathcal{P}(Y), \sqsubseteq_1') \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} (\mathcal{P}(\hat{Y}), \sqsubseteq_2')}{(\mathcal{P}(X \times Y), \sqsubseteq_1'') \underset{\alpha''}{\overset{\gamma''}{\leftrightarrows}} (\mathcal{P}(\hat{X} \times \hat{Y}), \sqsubseteq_2'')} \qquad \text{(function)}$$

Fig. 2. Structural inference rules for generating an abstract-state space. Once a designer specifies a Galois connection over the leaves of the concrete state-space, these inference rules construct an abstract state-space and corresponding abstraction/concretization functions.

5.4 Synthesizing an abstract interpretation for CPS (Option 1)

Returning to the CPS semantics in which we snipped the $Env \to Clo$ edge and defining an extraction function on addresses $\eta : Addr \to \widehat{Addr}$, we can recursively apply inference rules for Galois connections that lead us to a Galois connection $(\mathcal{P}(\Sigma), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{P}(\hat{\Sigma}), \sqsubseteq_{\mathcal{P}(\hat{\Sigma})})$ for the top-level state-space:

$$\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Call} \times \widehat{Env} \times \widehat{Store} \\
\hat{\rho} \in \widehat{Env} &= \mathsf{Var} \rightharpoonup \widehat{Addr} \\
\widehat{clo} \in \widehat{Clo} &= \mathsf{Lam} \times \widehat{Env} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \rightharpoonup \widehat{Clo} \\
\hat{a} \in \widehat{Addr} &\text{ is a finite set of addresses.}
\end{aligned}$$

The function $\alpha : \mathcal{P}(\Sigma) \to \mathcal{P}(\hat{\Sigma})$ encodes the synthesized abstraction map:

$$\begin{aligned}
\alpha\{(call, \rho, \sigma)\} &= \{(call, \alpha(\rho), \alpha(\sigma))\} \\
\alpha(\rho) &= \lambda v.\alpha(\rho(v)) \\
\alpha(\sigma) &= \lambda \hat{a}. \bigsqcup_{\alpha(a)=\hat{a}} \alpha(\sigma(a)) \\
\alpha(lam, \rho) &= \{(lam, \alpha(\rho))\} \\
\alpha(a) &= \eta(a).
\end{aligned}$$

Because we have a Galois connection, we can calculate an approximation of the "optimal" abstract transition relation, $(\leadsto) \subseteq \hat{\Sigma} \times \hat{\Sigma}$:

$$\overbrace{(\llbracket (f\ e_1 \ldots e_n) \rrbracket, \hat{\rho}, \hat{\sigma})}^{\hat{\varsigma}} \leadsto \overbrace{(call, \hat{\rho}'', \hat{\sigma}')}^{\hat{\varsigma}'}, \text{ where}$$

$$\begin{aligned}
(lam, \hat{\rho}') &\in \hat{\mathcal{E}}(f, \hat{\rho}, \hat{\sigma}) \\
lam &= \llbracket (\lambda\ (v_1 \ldots v_n)\ call) \rrbracket \\
\hat{a}_i &= \widehat{alloc}(v_i, \hat{\varsigma}) \\
\hat{\rho}'' &= \hat{\rho}'[v_i \mapsto \hat{a}_i] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{E}}(e_i, \hat{\rho}, \hat{\sigma})],
\end{aligned}$$

where the argument evaluator $\hat{\mathcal{E}} : \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \rightharpoonup \widehat{Clo}$ evaluates an expression in the context of an environment and a store to return a value:

$$\begin{aligned}
\hat{\mathcal{E}}(v, \hat{\rho}, \hat{\sigma}) &= \hat{\sigma}(\hat{\rho}(v)) \\
\hat{\mathcal{E}}(lam, \hat{\rho}, \hat{\sigma}) &= \{(lam, \hat{\rho})\}.
\end{aligned}$$

We also introduced the abstract address-allocation function $\widehat{alloc} : \mathsf{Var} \times \hat{\Sigma} \to \widehat{Addr}$. (The concrete semantics selected addresses nondeterministically from outside the domain of the store.) According to Might's *A Posteriori* Soundness Theorem [9], *any* abstract address allocator leads to a sound abstract interpretation.

Example 2. For example, a simple, monovariant address allocator chooses the variable itself for the abstract address:

$$\begin{aligned}\widehat{Addr} &= \mathsf{Var} \\ \widehat{alloc}(v, \hat{\varsigma}) &= v,\end{aligned}$$

which leads to an abstract interpretive formulation of 0CFA.

5.5 Synthesizing an abstract interpretation for CPS (Option 2)

Recall that the other option for eliminating recursion is to snip the $Clo \to Env$ edge. This snip leads to a family of analyses with a character unlike any in the published literature on higher-order flow analysis.

Snipping this edge and synthesizing an abstraction leads to the following abstract state-space:

$$\begin{aligned}\hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Call} \times \widehat{Env} \times \widehat{Store} \\ \hat{\rho} \in \widehat{Env} &= \mathsf{Var} \rightharpoonup \widehat{Clo} \\ \widehat{clo} \in \widehat{Clo} &= \mathsf{Lam} \times \widehat{Addr} \\ \hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \to \mathcal{P}\left(\widehat{Env}\right) \\ \hat{a} \in \widehat{Addr} &\text{ is an finite set of addresses,}\end{aligned}$$

and the following transition rule:

$$\overbrace{(\llbracket (f\ e_1 \ldots e_n) \rrbracket, \hat{\rho}, \hat{\sigma})}^{\hat{\varsigma}} \rightsquigarrow \overbrace{(call, \hat{\rho}'', \hat{\sigma}')}^{\hat{\varsigma}'}, \text{ where}$$

$$\begin{aligned}\hat{a} &= \widehat{alloc}(\hat{\varsigma}) \\ \hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{\rho}] \\ (lam, \hat{a}') &\in \hat{\mathcal{E}}(f, \hat{a}, \hat{\sigma}') \\ lam &= \llbracket (\lambda\ (v_1 \ldots v_n)\ call) \rrbracket \\ \widehat{clo}_i &= \hat{\mathcal{E}}(e_i, \hat{a}, \hat{\sigma}') \\ \hat{\rho}'' &= (\hat{\sigma}(\hat{a}'))[v_i \mapsto \widehat{clo}_i],\end{aligned}$$

where the argument evaluator $\hat{\mathcal{E}} : \mathsf{Exp} \times \widehat{Addr} \times \widehat{Store} \to \mathcal{P}(\widehat{Clo})$ evaluates an expression in the context of an environment's address and a store to return a value:

$$\begin{aligned}\hat{\mathcal{E}}(v, \hat{a}, \hat{\sigma}) &= \{\hat{\rho}(v) : \hat{\rho} \in \hat{\sigma}(a)\} \\ \hat{\mathcal{E}}(lam, \hat{a}, \hat{\sigma}) &= \{(lam, a)\}\,.\end{aligned}$$

6 Flow-sensitivity, field-sensitivity and context-sensitivity

We mentioned earlier that snipping different edges could lead to different knobs for tuning the precision of the analysis. Properties such as flow-, field- and context-sensitivity emerge as the result of extra snips in the original dependence graph, and their degree can be tuned by the extraction function required to form the Galois connection.

Flow-sensitivity Consider, for example, snipping the $\Sigma \to \mathsf{Call}$ edge in the CPS semantics. That is, instead of a state having the structure $\varsigma = (call, \ldots)$, it will have the structure $\varsigma' = (a_{call}, \ldots, \sigma)$, where $call = \sigma(a_{call})$. Thus, call sites become addressable values, and to abstract, one must define an extraction function. This extraction function on addresses of call sites creates a concrete leaf node that, under the second step, maps to "abstract call sites." If all concrete call sites abstract to the same abstract call site, *i.e.* $\eta(a_{call}) = \hat{a}_0$ for all call site addresses a_{call}, then the optimal analysis becomes completely flow-insensitive. If, on the other hand, the extraction function is the identity function, then the optimal analysis is completely flow-sensitive. The nature of the abstraction from concrete to abstract call sites precisely captures the flow-sensitivity of the resulting analysis.

Field-sensitivity In higher-order languages, environments play the role of structures. Thus, for CPS, field-sensitivity manifests as the degree to which variables in a given environment have the same abstract address. To create a Galois connection that tunes this parameter, we need only snip the $Env \to \mathsf{Var}$ edge in the concrete dependence graph. Once again, a singleton extraction map leads to field-insensitivity, and an identity extraction map leads to field-sensitivity.

Context-sensitivity and polyvariance The term *polyvariance* refers to the number of abstractions (variants) for a given variable (or allocation site). Monovariant analyses like 0CFA have only one abstract address for each variable. Typically, context-sensitivity determines polyvariance by carving up the abstract variants of a variable according the contexts in which it is bound. Thus, to tune polyvariance, snip the $Env \to Clo$ edge in the concrete state-space graph, and adjust the extraction function for the resulting Galois connection.

7 Related work

This work draws most directly on three lines of research: abstract interpretation [4], formal semantics [17] and Galois connections [5]. The programmatic transformation of formal semantics dates to work by Reynolds [15]. More recent work by Danvy *et al.* has shown that formal semantics are highly amenable to program transformations and that it is possible to automatically convert denotational semantics into operational semantics and *vice versa* [2, 3, 1, 6–8]. These techniques, combined with ours, should permit the mechanizable construction of static analyzers for a wider variety of formal semantics paradigms.

The Cousots' foundational work on Galois connections marks the earliest attempts to mechanize the process of constructing an abstract interpretation [5]. Given a Galois connection $X \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{X}$, it is possible to calculate the optimal abstract image of a concrete function $f : X \to X$ as $\hat{f} = \alpha \circ f \circ \gamma$. Our work advances the Cousots' original work by automating the construction of the Galois connection itself using inference rules. There have been additional attempts to automate parts of the process of constructing an abstraction; most recently, work by Qian *et al.* has focused on constructing minimal abstractions that lead to completeness [14].

Our running example on the abstraction of continuation-passing style lambda calculus is an instance of the long line of work on higher-order control-flow analysis [19]. The first family of analyses we derived corresponds to universal framework for k-CFA-like analyses [12]. The second family of analyses we derived is difficult to place in relation to existing analyses. To begin, it is the only analysis which does not abstract the range of environments. This gives it the unique feature that variable look-up in such an analysis yields exactly one abstract closure. It also opens up the prospect of using techniques such as abstract counting directly on environment addresses in order to perform must-alias analysis [10, 11].

8 Summary and conclusion

We have presented a two-step method for converting a small-step concrete semantics into an abstract interpretation. The first step eliminates recursive structure from the concrete state-space by snipping edges in the dependence graph of the concrete state-space; the second step trickles abstraction up the leaves of the newly re-factored concrete state-space. Inference rules over structural Galois connections synthesize the abstract state-space, and a Galois connection between concrete and abstract states at the same time. The synthesized Galois connection also determines the optimal abstract interpretation. By snipping additional edges in the concrete dependence graph, these snips turn into knobs for tuning flow-, field- and context-sensitivity under abstraction. The immediate payoff of this method in our work was (1) a re-affirmation that k-CFA is, in some sense a fundamental technique, and (2) a new family of analyses based on a novel abstraction of environments.

References

1. *A functional correspondence between evaluators and abstract machines* (2003), ACM Press.
2. Ager, M., Danvy, O., and Midtgaard, J. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science 342*, 1 (September 2005), 149–172.
3. Ager, M. S., Danvy, O., and Midtgaard, J. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters 90*, 5 (June 2004), 223–232.

4. Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages* (New York, NY, USA, 1977), ACM Press, pp. 238–252.
5. Cousot, P., and Cousot, R. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1979), ACM Press, pp. 269–282.
6. Danvy, O., and Millikin, K. A rational deconstruction of landin's secd machine with the j operator. *Logical Methods in Computer Science 4*, 4 (November 2008).
7. Danvy, O., and Millikin, K. Refunctionalization at work. *Science of Computer Programming 74*, 8 (June 2009), 534–549.
8. Midtgaard, J. *Transformation, Analysis, and Interpretation of Higher-Order Procedural Programs.* PhD thesis, University of Aarhus, 2007.
9. Might, M., and Manolios, P. A posteriori soundness for non-deterministic abstract interpretations. In *VMCAI '09: Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 260–274.
10. Might, M., and Shivers, O. Improving flow analyses via γcfa: Abstract garbage collection and counting. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2006), ACM, pp. 13–25.
11. Might, M., and Shivers, O. Exploiting reachability and cardinality in higher-order flow analysis. *Journal of Functional Programming 18*, Special Double Issue 5-6 (2008), 821–864.
12. Nielson, F., and Nielson, H. R. Infinitary control flow analysis: a collecting semantics for closure analysis. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1997), ACM, pp. 332–345.
13. Nielson, F., Nielson, H. R., and Hankin, C. *Principles of Program Analysis*, corrected ed. Springer, October 1999.
14. Qian, J., Zhao, L., Cai, G., and Gu, T. Automatic construction of complete abstraction by abstract interpretation. In *ICIS '09: Proceedings of the 2009 Eigth IEEE/ACIS International Conference on Computer and Information Science* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 927–932.
15. Reynolds, J. C. Definitional interpreters for higher-order programming languages. In *ACM 1972: Proceedings of the ACM Annual Conference* (New York, NY, USA, 1972), ACM, pp. 717–740.
16. Schmidt, D. A. Abstract interpretation of small-step semantics. In *Selected papers from the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages* (London, UK, 1997), Springer-Verlag, pp. 76–99.
17. Scott, D., and Strachey, C. *Towards a formal semantics.* 1966, pp. 197–220.
18. Shivers, O. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (New York, NY, USA, July 1988), vol. 23, ACM, pp. 164–174.
19. Shivers, O. G. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, 1991.
20. Wand, M., and Siveroni, I. Constraint systems for useless variable elimination. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1999), ACM, pp. 291–302.