Natural-Semantics-Based Abstract Interpretation (Preliminary Version)

David A. Schmidt
Department of Computing and Information Sciences
Kansas State University *

Abstract

The original formulation of abstract interpretation (*a.i.*) [5] demonstrated clearly that *a.i.* is a formal-semantics-based methodology for deriving a provably correct, convergent, *canonical* iterative data flow analysis from a standard semantics of a programming language. But subsequent research in *a.i.* has obscured the methodology of the topic. For example, the recent slew of papers on closures analysis [2, 3, 17, 18, 21, 37, 39, 40, 41, 42, 43] mix implementation optimizations with specifications and leave unclear exactly what closures analysis is. In this paper, we reexamine the principles of *a.i.* and reformulate the topic on a foundation of coinductively defined natural semantics. We aim to demonstrate that the intensional and compositional aspects of natural semantics make it an ideal vehicle for formulating abstract interpretations of problems while preserving the essential characteristics of the subject.

1 Background

In data flow analysis (*d.f.a.*) [1, 15, 23, 24, 30], one represents a flow chart program as a set of simultaneous equations that define sets of values (or properties) at the arcs of the chart. The solution of the equations computes the sets of values (or properties) that "flowed" along the arcs during the flow chart's computation. The intuitions of *d.f.a.* are strong, but correctness proofs can be difficult. Perhaps as a consequence, research in *d.f.a.* focussed upon efficiency of solutions and application of the results to code improvement.

A major breakthrough was the abstract interpretation (*a.i.*) method, defined in almost complete maturity, by Cousot and Cousot [5]. The key ideas are:

1. The execution of a flow chart with run-time values is the *concrete* or *standard semantics*. When one collects into sets the values that traversed the flowchart's arcs, one obtains the *collecting semantics*.[1]

*Manhattan, KS 66506 USA. `schmidt@cis.ksu.edu`. Partially supported by NSF CCR-9302962 and ONR N00014-94-1-0866.

[1] In [5], this is called the *static semantics*.

2. The execution of the flow chart with abstract values ("properties") is the *abstract semantics*. When one collects the abstract values that traversed the flowchart's arcs, one obtains a *d.f.a.*

3. The soundness of the abstract semantics to the standard semantics can be rigorously proved by relating concrete and abstract value sets via a Galois connection [6, 26]. This result is "lifted" to a proof of soundness of the *d.f.a.* to the collecting semantics.

4. A *d.f.a.* must be finitely convergent, so restrictions must be enforced upon it (e.g., a monotone analysis with values partially ordered with a finite chain property or use of a widening-narrowing technique).

Cousot and Cousot [5] make clear that that *a.i.* is a *canonical*, provably correct formalization of *d.f.a.* —a major accomplishment.

A subsequent advance was the formulation of *a.i.* in a compositional manner via denotational semantics definitions[10, 34, 35]: flow chart equations were replaced by structurally recursive equations, and abstract value sets were formalized as Scott-domains. These techniques were elegant, and they opened the door to analysis of higher-order programs (e.g., strictness analysis [4, 31]), but there was a price to be paid: a fundamentally intensional analysis was made into an extensional one. This impacts the collecting semantics, which gathers the intensional information; to encode a collecting semantics, the denotational semantics definition must be augmented by "Label" or "Occurrence" arguments, which label "program points" [10]. Also, a store-like "cache" argument is added to collect the values that were supplied as arguments to the functions labelled by the program points [16].

The rewriting of the standard semantic definition into the collecting semantics definition breaks the relationship between the two, and now one must prove that the collecting semantics is correct with respect to the standard semantics, but it is unclear how one does this precisely.[2]

A variant of the above is the minimal function graph semantics of Jones, Mycroft, and Rosendahl [20, 21]: a program is a set of named function definitions, and given an input value, one calculates the semantics of the functions as the collection of smallest function graphs that can map the input to an output. When one views the function names as labels, then the semantics just described is the collecting semantics.

Another complication is convergence checking. Since a denotational semantics is not necessarily a tail-recursive set of equations (like a flowchart), one cannot see when convergence of the *d.f.a.* is reached [29].

One might summarize the above narrative by stating that *a.i.* does not generalize from flowchart programs to structurally recursive ones as easily as one would like. But the problem is not the structurally recursive programs—it is the loss of intensionality in the denotational-semantics-style definitions. For this reason, we redevelop *a.i.* based on a coinductively defined natural semantics.[3] The key aspects of *a.i.* will be recovered:

1. A program's standard semantics is exactly its natural semantics tree; the collecting semantics is the union of the information at the nodes of the natural semantics tree.

[2] In [34], Nielson notes this issue and supplies a proof of extensional (and not intensional) correctness.

[3] In Milner and Tofte [27] this is called "relational semantics."

2. A program's abstract semantics is the natural semantics calculated with abstract values; the *d.f.a.* is the join of the information at the nodes of the abstract semantics tree.

3. Safety is proved by a Galois connection (or logical relation [32, 36, 38]) relating concrete to abstract values; this "lifts" to a safety proof of the collecting semantics to the *d.f.a.*

4. Convergence checking is done by unfolding the abstract semantics tree until each path terminates or contains a repeating node.

Once developed, we apply the methodology to a first-order signs analysis and a (higher-order) closures analysis.

2 Inductively and Coinductively Defined Sets

The usual, inductively defined version of natural semantics [22] can express only convergent computations, therefore we must give a coinductive formulation for our work. The following material is summarized from Cousot and Cousot [7].

We begin with the classical definition of inductive definition. Let $\mathcal{U}$ be a universe of terms, and let $F\colon \wp(\mathcal{U}) \to \wp(\mathcal{U})$ be continuous[4] with respect to the complete lattice $\langle \wp(\mathcal{U}), \subseteq \rangle$. The *set defined inductively by* F is $\mathit{lfp}F = \bigcup_{i \geq 0} S_i$, where $S_0 = \{\}$ and $S_{i+1} = F(S_i)$. Note also that $\mathit{lfp}F = \bigcap\{S' \mid \mathit{closed}_F S'\}$, where $\mathit{closed}_F S'$ iff $F(S') \subseteq S'$. That is, $\mathit{lfp}F$ is the smallest closed set. The latter definition gives a standard reasoning technique, *fixed point induction*: to prove $\mathit{lfp}F \subseteq P$, that is, every element of $\mathit{lfp}F$ has property P, it suffices to find a set $S' \subseteq P$ such that $\mathit{closed}_F S'$. When F is defined from a BNF rule (see below), then proving $\mathit{closed}_F P$ is a *structural induction* proof.

When the above definitions are dualized, we obtain coinduction: for $\mathcal{U}$ and F as above, the *set defined coinductively by* F is $\mathit{gfp}F = \bigcap_{i \geq 0} T_i$, where $T_0 = \mathcal{U}$ and $T_{i+1} = F(T_i)$. Also, $\mathit{gfp}F = \bigcup\{T' \mid \mathit{dense}_F T'\}$, where $\mathit{dense}_F T'$ iff $T' \subseteq F(T')$. That is, $\mathit{gfp}F$ is the largest dense set. This gives the reasoning technique of *fixed point coinduction*: to prove $P \subseteq \mathit{gfp}F$, it suffices to find a set, P', such that $P \subseteq P'$ and $\mathit{dense}_F P'$.

Here are brief examples. Let $\mathcal{U}$ be a universe of strings of at most countably infinite (ω-) length; this BNF rule:

$$V \ ::= \ 0 \ \mid \ 1V$$

generates the continuous functional $\bar{V}(S) = \{0\} \cup \{1s \mid s \in S\}$; we obtain $\mathit{lfp}\bar{V} = \{1^n 0 \mid n \geq 0\}$, whereas $\mathit{gfp}\bar{V} = \mathit{lfp}\bar{V} \cup \{1^\omega\}$.

In the sequel, it is helpful to think of strings as trees with a single path; in the case of $\mathit{gfp}\bar{V}$, we see that set T_i contains those trees that are verified well formed as far as depth i. Another view is this: when calculating $\mathit{lfp}\bar{V}$, S_i contains those trees that are certainly members of $\mathit{lfp}\bar{V}$; in contrast, T_i contains those trees that are not yet excluded from membership in $\mathit{gfp}\bar{V}$.

Say that we wish to prove that all strings in $\mathit{lfp}\bar{V}$ are finite: by fixed-point induction, we need only show that the set $\mathit{is\_finite} \subseteq \mathcal{U}$ is closed: $\bar{V}(\mathit{is\_finite}) \subseteq \mathit{is\_finite}$. This is the usual structural induction proof. A typical application of fixed-point coinduction takes more

[4] A monotone function would suffice, but continuity ensures fixed point convergence by the first limit ordinal.

$$\pi \vdash b : \iota \qquad\qquad \pi \vdash x : \tau \ \text{ if } (x{:}\,\tau) \in \pi$$

$$\frac{\pi \vdash e_1 : int \quad \pi \vdash e_2 : int}{\pi \vdash e_1 + e_2 : int} \qquad\qquad \frac{\pi \vdash e_1 : bool \quad \pi \vdash e_2 : \tau \quad \pi \vdash e_3 : \tau}{\pi \vdash \texttt{if}\, e_1\, e_2\, e_3 : \tau}$$

$$\frac{\{x{:}\,\tau_1\} \oplus \pi \vdash e : \tau_2}{\pi \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad\qquad \frac{\pi \vdash e_1 : \tau_1 \to \tau_2 \quad \pi \vdash e_2 : \tau_1}{\pi \vdash e_1\, e_2 : \tau_2}$$

$$\frac{\{f{:}\,\tau_1 \to \tau_2\} \oplus \{x{:}\,\tau_1\} \oplus \pi \vdash e : \tau_2}{\pi \vdash \texttt{fix}\, f\, x.e : \tau_1 \to \tau_2}$$

Figure 1: Source Language Syntax

work: say that we wish to show, for all strings (trees) in $gfp\bar{V}$, that no 1 follows a 0. Define this predicate:

$$ok(s) \text{ iff } s = 0 \text{ or } s = 1 \text{ or } (s = 1s' \text{ and } ok(s'))$$

This is a circular predicate, so consider the corresponding functional: $\bar{ok}(P) = \{s \mid s = 0 \text{ or } s = 1 \text{ or } (s = 1s' \text{ and } s' \in P)\}$ and define $Ok = gfp\ \bar{ok}$. (This ensures that $1^\omega \in Ok$.) To prove $gfp\bar{V}$, it suffices to prove $dense_{\bar{ok}}\ gfp\bar{V}$: $gfp\bar{V} \subseteq \bar{ok}(gfp\bar{V})$.

For the remainder of this paper, we use a universe, $\mathcal{U}$, of finitely branching trees of at most countably infinite (ω-) depth[11, 13]. We generate such trees with inference rule schemes. For example, the set of rule schemes, $\Phi = \{\vdash 0, \frac{\vdash n}{\vdash 1}\}$, generate trees with a single path whose nodes are labelled by "$\vdash n$", $n \in \{0,1\}$. An example is $\begin{array}{c}\vdash 0\\ | \\ \vdash 1\\ | \\ \vdash 1\end{array}$ and the functional corresponding to Φ is $\bar{\Phi}(T) = \{\vdash 0\} \cup \{\overset{t}{\vdash 1} \mid t \in T\}$.

Therefore, $lfp\bar{\Phi}$ is $lfp\bar{V}$ in tree form; $gfp\bar{\Phi}$ is similarly $gfp\bar{V}$.

3 Natural Semantics for a Higher-Order Functional Language

Our example language is a simply-typed, call-by-value lambda calculus. Figure 1 gives the syntax for the source language, and Figure 2 shows the concrete value sets to be used in the natural semantics. The set of source programs is defined as the usual inductive set, but the *Env* and *Value* sets are defined coinductively, so we obtain "infinite closures," which we will use to define the semantics of the `fix` operator. It is helpful to view the values and environments as trees whose nodes are labelled with symbols; here is Figure 2 redrawn to generate trees:

$$v \ ::= \ b \ \mid \ \langle x, e, \overset{\rho}{\ } \rangle \ \mid \ \bot \qquad\qquad \rho \ ::= \ \{i_1 = \overset{v_1}{\ }, i_2 = \overset{v_2}{\ }, \ldots, i_n = \overset{v_n}{\ }\}$$

We write $root(t)$ to be the symbols labelling the root node of t, and assuming an indexing scheme for the nodes of t, we write $node(t_i)$ to denote the symbols that label node i of tree

$$
\begin{array}{lll}
x \in \mathit{Identifier} & & \\
e \in \mathit{Expression} & & \\
b \in \mathit{Base\text{-}constant} = \mathit{Int} \cup \mathit{Bool} & & \\
v \in \mathit{Value} & v \ ::= \ b \ \mid \ \langle x, e, \rho \rangle \ \mid \ \perp & \\
\rho \in \mathit{Env} & \rho \ ::= \ \{i = v_i\}_{i \in Id}, \ Id \subseteq \mathit{Identifier} &
\end{array}
$$

Figure 2: Value Sets for Semantics

t. Next, we define the binary relations $\sqsubseteq_{Value}$ and $\sqsubseteq_{Env}$ coinductively, that is, as the largest binary relations satisfying these properties:

- $\perp \sqsubseteq_{Value} v$ iff $v \in \mathit{Value}$
- $b_1 \sqsubseteq_{Value} b_2$ iff $b_1, b_2 \in \mathit{Base\text{-}constant}$ *and* $b_1 = b_2$
- $\langle x_1, e_1, \rho_1 \rangle \sqsubseteq_{Value} \langle x_2, e_2, \rho_2 \rangle$ iff $x_1 = x_2, e_1 = e_2$, and $\rho_1 \sqsubseteq_{Env} \rho_2$
- $\rho_1 \sqsubseteq_{Env} \rho_2$ iff $\mathit{domain}(\rho_1) = \mathit{domain}(\rho_2)$ and for all $i \in \mathit{domain}(\rho_1), \rho_1(i) \sqsubseteq_{Value} \rho_2(i)$

We take $\sqsubseteq_{Value}$ and $\sqsubseteq_{Env}$ to be the largest relations (rather than the smallest), so that infinite trees are included in the ordering. With fixed-point coinduction, we can prove:

- $\sqsubseteq_{Value}$ and $\sqsubseteq_{Env}$ are (pointed) ω-complete partial orderings (*c.p.o.s*).
- for $D \in \{\mathit{Value}, \mathit{Env}\}$, $a, b \in D$, $a \sqsubseteq_D b$ iff for all nodes, i, of a, either $\mathit{node}(a_i) = \perp$ or $\mathit{node}(a_i) = \mathit{node}(b_i)$. That is, b has the same branching structure as a and has the same nodes as a up to a's $\perp$ leaves.

Since *Value* is a c.p.o., we can define infinite closures like $c_\infty = \langle x, e, \rho \oplus (f = c_\infty) \rangle$ by the usual least fixed point interpretation: $c_\infty = \bigsqcup_{i \geq 0} c_i$, where $c_0 = \perp$, and $c_{i+1} = \langle x, e, \rho \oplus (f = c_i) \rangle$.

4 Natural Semantics Trees

Figure 3 shows the natural semantics for the source language. The rule schemes are as expected, and as noted earlier `fix` is modelled by an infinite closure. We intend to define the language of natural semantics trees as the set coinductively generated from the rules in Figure 3, but there is a bit of groundwork to do first. Let the Universe, $\mathcal{U}$, be the set of ω-depth, finitely branching trees whose nodes are labelled by "$\rho \vdash e \Downarrow v$" or by "$\Omega$". (Think of Ω as "undefined.") Define a binary relation, $\preceq \subseteq \mathcal{U} \times \mathcal{U}$, as the largest binary relation satisfying the following properties:

- $t_1 \preceq t_2$ if $t_1 = \Omega$
- $t_1 \preceq t_2$ if $\mathit{root}(t_1) \sqsubseteq \mathit{root}(t_2)$, $\mathit{number\_of\_subtrees}(t_1) = \mathit{number\_of\_subtrees}(t_2)$, and for all subtrees, i, of t_1, $t_{1i} \preceq t_{2i}$, where $\mathit{root}(t_1) \sqsubseteq \mathit{root}(t_2)$ is defined as: $\rho_1 \vdash e_1 \Downarrow v_1 \sqsubseteq \rho_2 \vdash e_2 \Downarrow v_2$ iff $\rho_1 \sqsubseteq_{Env} \rho_2, e_1 = e_2$, and $v_1 \sqsubseteq_{Value} v_2$.

$$\rho \vdash b \Downarrow b \qquad \frac{\rho \vdash e_1 \Downarrow v_1 \quad \rho \vdash e_2 \Downarrow v_2}{\rho \vdash e_1 + e_2 \Downarrow plus(v_1, v_2)}$$

$$\frac{\rho \vdash e_1 \Downarrow \text{true} \quad \rho \vdash e_2 \Downarrow v}{\rho \vdash \texttt{if}\, e_1\, e_2\, e_3 \Downarrow v} \qquad \frac{\rho \vdash e_1 \Downarrow \text{false} \quad \rho \vdash e_3 \Downarrow v}{\rho \vdash \texttt{if}\, e_1\, e_2\, e_3 \Downarrow v}$$

$$\rho \vdash x \Downarrow v \;\; \text{if}\, (x = v) \in \rho \qquad \rho \vdash \lambda x.e \Downarrow \langle x, e, \rho\rangle$$

$$\frac{\rho \vdash e_1 \Downarrow \langle x', e', \rho'\rangle \quad \rho \vdash e_2 \Downarrow v' \quad \rho' \oplus \{x' = v'\} \vdash e' \Downarrow v}{\rho \vdash e_1\, e_2 \Downarrow v}$$

$$\rho \vdash \texttt{fix}\, f\, x.e \Downarrow c_\infty \text{ where } c_\infty = \langle x, e, \rho \oplus \{f = c_\infty\}\rangle$$

Figure 3: Natural Semantics

One can use fixed-point coinduction to prove that $\preceq$ is a *c.p.o.* , and $t_1 \preceq t_2$ exactly when t_2 has the same branching structure as t_1, and for all nodes, i of t_1, $t_{1i} = \Omega$ or $node(t_{1i}) \sqsubseteq node(t_{2i})$.

Figure 3 defines a set, Φ, of rule schemes that have form:

$$\frac{\{\rho_i \vdash e_i \Downarrow v_i\}_{i \in 1..n}}{\rho \vdash op(e_i)_{i \in 1..n} \Downarrow f(v_i)_{i \in 1..n}}$$

It is helpful to view Φ as an attribute grammar, where the ρ_i-environments are inherited attibutes, and the v_i-values are synthesized ones. We require that Φ be an *L-attributed attribute grammar*[25]: a v_i can appear within a ρ_j only if $j > i$. Rule schemes are instantiated in the usual way, but $\perp$ can instantiate a v_i and Ω can instantiate an antecedent scheme, $\rho_i \vdash e_i \Downarrow v_i$, subject to these *strictness* conditions:

- $f(v_i)_{i \in 1..n} = \perp$ if an antecedent scheme, $\rho_k \vdash e_k \Downarrow v_k$, is instantiated by Ω.
- an antecedent scheme, $\rho_k \vdash e_k \Downarrow v_k$, is instantiated by Ω if ρ_k contains a v_j and the corresponding scheme, $\rho_j \vdash e_j \Downarrow v_j$, that defines v_j is itself instantiated to Ω, or $v_j = \perp$.

The intuition is that $\perp$ represents no output and Ω represents a missing computation; a missing computation causes other, dependent computations to be missing and causes an ultimate result of $\perp$.

Given rule scheme set, Φ, we define the corresponding functional, $\bar{\Phi}$, in the fashion seen in Section 2, and take $gfp\bar{\Phi}$ to be the well-formed derivations of Φ. $gfp\bar{\Phi}$ includes the usual, finite-depth trees that one obtains for a traditional natural semantics, but ω-depth trees also appear—indeed there are more than what we require. For example, for every $\rho \in Env$ and every $v \in Value$, there is an infinite, well-formed tree with root $\rho \vdash (\texttt{fix}\, f\, x.f\, x)0 \Downarrow v$. Of course, we desire the tree where v is $\perp$, and this tree is the least (with respect to $\preceq$) of the well-formed trees. A more direct example is seen from the rule

$$
\begin{aligned}
&\mathcal{E}\ x\ \rho = \rho \vdash x \Downarrow v,\ \text{if } (x = v) \in \rho \\
&\mathcal{E}\ (\lambda x.e)\ \rho = \rho \vdash \lambda x.e \Downarrow \langle x, e, \rho\rangle \\
&\mathcal{E}\ (e_1\, e_2)\ \rho = \ \ \text{let } t_1 = \mathcal{E}\ e_1\ \rho \\
&\qquad\qquad\qquad\ \ \text{let } t_2 = \mathcal{E}\ e_2\ \rho \\
&\qquad\qquad\qquad\ \ \text{let } t_3 = \mathcal{E}\ e'\ (\rho' \oplus \{x' = v'\}) \\
&\qquad\qquad\qquad\ \ \text{in } (\rho \vdash e_1\, e_2 \Downarrow \mathit{val}(\mathit{root}\, t_3))@\{t_1, t_2, t_3\} \\
&\qquad\qquad\qquad\quad \text{where } \mathit{val}(\rho \vdash e \Downarrow v) = v \\
&\qquad\qquad\qquad\quad n@\{t_1, ..., t_n\} \text{ is the tree with root } n \text{ and subtrees } t_1, ..., t_n
\end{aligned}
$$

Figure 4: Interpreter Rule Schemes

$$\frac{\rho \vdash \mathit{loop} \Downarrow v}{\rho \vdash \mathit{loop} \Downarrow v} \quad \text{where, for all } v_0, \text{ we have the tree:} \quad \begin{array}{c} \vdots \\ \rho \vdash \mathit{loop} \Downarrow v_0 \\ | \\ \rho \vdash \mathit{loop} \Downarrow v_0 \\ | \\ \rho \vdash \mathit{loop} \Downarrow v_0 \end{array}$$

Thus, we legislate that the *natural semantics of* ρ_0, e_0, is the least $t \in \mathit{gfp}\bar{\Phi}$ such that $\mathit{root}(t) = \rho_0 \vdash e_0 \Downarrow v$, for some $v \in \mathit{Value}$.

Every ρ_0, e_0 pair has a least tree, and we can generate it by using the rule set, Φ, as if it were an "interpreter" coded in Typol [9] or Prolog. A formalization of this idea in equational, least-fixed-point semantics is to rewrite a rule scheme of the form

$$\frac{\{\rho_i \vdash e_i \Downarrow v_i\}_{i \in 1..n}}{\rho \vdash \mathit{op}(e_i)_{i \in 1..n} \Downarrow f(v_i)_{i \in 1..n}}$$

into this recursive equation scheme:

$$
\begin{aligned}
\mathcal{E}\ (\mathit{op}(e_i)_{i \in 1..n})\ \rho = \ &\text{let } \{\mathit{subtree}_i = \mathcal{E}\ e_i\ \rho_i\}_{i \in 1..n} \\
&\text{in } (\ \rho \vdash \mathit{op}(e_i)_{i \in 1..n} \Downarrow f(\mathit{val}(\mathit{root}(\mathit{subtree}_i)))_{i \in 1..n})\ @\ \{\mathit{subtree}_i\}_{i \in 1..n} \\
&\text{where } \mathit{val}(\Omega) = \bot \text{ and } \mathit{val}(\rho \vdash e \Downarrow v) = v \\
&\text{and } n\ @\ \{t_1, ..., t_n\} \text{ is the tree with root } n \text{ and subtrees } t_1, ..., t_n
\end{aligned}
$$

Figure 4 shows several of the equation schemes for the rule schemes in Figure 3.[5] It is crucial that the equation schemes are continuous with respect to the partial ordering on the *Env* set, and this is the case for the schemes in Figure 4. The equation schemes define a recursive function, $\mathcal{E}\colon \mathit{Expression} \to \mathit{Env} \to \mathcal{U}$, so let $\tilde{\Phi}$ be the corresponding functional. Then, the interpreter is defined $\mathcal{E} = \mathit{lfp}\tilde{\Phi} = \bigsqcup_{i \geq 0} \mathcal{E}_i$, where $\mathcal{E}_0 = \lambda e.\lambda \rho.\Omega$ and $\mathcal{E}_{i+1} = \tilde{\Phi}(\mathcal{E}_i)$. Proofs by induction on i let us prove that $\mathcal{E}$ has arity $\mathit{Expression} \to \mathit{Env} \to \mathit{gfp}\bar{\Phi}$ and that $\mathcal{E}\ e_0\ \rho_0$ is the least tree with root $\rho_0 \vdash e_0 \Downarrow v$.

At this point, one might introduce a call-by-value denotational semantics for the language and prove that the natural semantics is sound and adequate with respect to it. These steps

[5] Note that the two rules for `if` in Figure 3 must be merged into one to use this naive translation of rule schemes into recursive equations.

are straightforward: soundness and adequacy are proved in the standard way [14] with an inclusive logical relation. There is a similar development for a call-by-name version of the language.

5 Collecting Semantics

Since a natural semantics tree is essentially an "unfolded flowchart computation," it is easy to define its collecting semantics. As usual, we assume that the source program has labels attached to its syntax phrases,[6] and from here on, we write $l{:}\, e$ to represent that label $l \in \mathit{Label}$ is attached to phrase e.

The collecting semantics of a natural semantics tree, t, is simply

$$coll_{\sqcup}(t) = \lambda l{:}\, \mathit{Label}.\ \bigsqcup\{(\rho, v) \ \mid\ \rho \vdash l{:}\, e \Downarrow v \text{ is a node of } t\}$$

In the usual case, the join operation, $\bigsqcup$, is set union.

6 Abstract Domains and Safety

The central concept of abstract interpretation is that a language's *concrete*, run-time value set, *Value*, can be replaced by an *abstract*, property set, *Abstract_value*, thus defining the program's abstract semantics. A program's abstract semantics must predict the program's concrete semantics, so a formal relationship must be established between concrete and abstract sets. This relationship can be defined as a Galois connection [6, 26] or as a logical-style relation [32, 36, 38]. These concepts are well established, so we summarize the key points, based primarily on Nielson [36].

Let C be a partially-ordered set of concrete values, and let A be a complete lattice of abstract values. We define the relationship between C and A by a function $\beta : C \to A$, which maps each concrete value to the element in A that best describes it. (Example: $C = \mathit{Integer}$, $A = \{\bot, \mathit{negative}, \mathit{zero}, \mathit{positive}, \top\}$, so $\beta(1) = \mathit{positive}$.) Next, define the binary relation $c\ \hat{\beta}\ a$ iff $\beta(c) \sqsubseteq a$.

Here are some properties of binary relations (cf. [32]): for $\mathit{rel} \subseteq P \times Q$, $p \in P$, $q \in Q$, rel is

- *L-closed* iff $p\ \mathit{rel}\ q$ and $p' \sqsubseteq_P p$ imply $p'\ \mathit{rel}\ q$
- *U-closed* iff $p\ \mathit{rel}\ q$ and $q \sqsubseteq_Q q'$ imply $p\ \mathit{rel}\ q'$
- *G-closed* iff $p\ \mathit{rel} \sqcap\{q \ \mid\ p\ \mathit{rel}\ q\}$
- *inclusive* iff for all ω-chains $\{p_i\}_{i\geq 0} \subseteq P$, $\{q_i\}_{i\geq 0} \subseteq Q$, if for all $i \geq 0$, $p_i\ \mathit{rel}\ q_i$ then $\bigsqcup_{i\geq 0} p_i\ \mathit{rel} \bigsqcup_{i\geq 0} q_i$.

We have the following properties about $\hat{\beta} \subseteq C \times A$:

- $\hat{\beta}$ is GU-closed

[6] Or else we use the node addresses in the source program's parse tree as the labels.

- if β is monotone, then $\hat{\beta}$ is L-closed
- if β is continuous, then $\hat{\beta}$ is inclusive

Because $\hat{\beta}$ is GU-closed, it is easy to prove that the following functions define a Galois connection between $\wp(C)$[7] and A:[8]

- $\gamma : A \rightarrow \wp(C)$ is $\gamma(a) = \{c \mid c\ \hat{\beta}\ a\}$
- $\alpha : \wp(C) \rightarrow A$ is $\alpha(S) = \bigsqcup_{c \in S} \beta(c)$

Recall that γ maps an abstract value to the set of concrete values that it represents (that it "predicts"), and α maps a set of concrete values to the single abstract value that best describes the set. In practice, the γ function is crucial, because it lets us "decode" the results of an abstract interpretation; α "encodes" concrete values into properties; and the Galois connection criterion ensures that the encoding and decoding are consistent and the encoding is as precise as possible.

A convenient reformulation of the above is to begin with a binary relation, $rel \subseteq C \times A$, which defines which concrete values have which properties. Let $\beta' : C \rightarrow A$ be $\beta'(c) = \sqcap\{a \mid c\ rel\ a\}$; we have the following properties:

- if *rel* is GU-closed, then $\hat{\beta}' = rel$
- if *rel* is L-closed, then β' is monotone
- if *rel* is inclusive and GL-closed, then β' is continuous.

It is possible to work usefully with an inclusive, U-closed *rel* that is not L- or G-closed: define $\gamma : A \rightarrow \wp(C)$ as above, but define $\alpha : \wp(C) \rightarrow A$ as $\alpha(S) = \bigsqcup_{c \in S} \delta c$, where for all $c \in S$, $c\ rel\ \delta c$. That is, α is a safe but not necessarily precise-as-possible encoding of S (cf. *semi-adjoined functions* [33]). Further, A need not be a complete lattice, but this depends upon which joins in A are required (cf. uses of α).

7 Abstract Semantics Trees

Once the concrete domains are replaced by abstract ones, the definition of the set of well-formed abstract semantics trees goes basically as before: the universe, $\mathcal{U}_A$, is again the set of finitely branching trees of at most ω-depth, plus a new tree, Δ (read "Top"). An ordering, $\preceq \subseteq \mathcal{U}_A \times \mathcal{U}_A$, is defined to reflect that an abstract tree might investigate multiple possible computation sequences: for $t, u \in \mathcal{U}_A$,

$$\begin{aligned} & t \preceq u \text{ iff } t = \Omega \text{ orelse } u = \Delta \\ & \quad \text{orelse } root(t) \sqsubseteq root(u), \text{ and for all subtrees, } i, \text{ of } t, t_i \preceq u_i \end{aligned}$$[9]

[7] The powerset operator, $\wp$, is an ordinary powerset construction (an "approximation ordering" [8]) and not a powerdomain (a "computation ordering").

[8] A pair of maps, $f : A \rightarrow B$, $g : B \rightarrow A$, form a Galois connection iff for all $a \in A$, $b \in B$, $f(a) \sqsubseteq_B b$ iff $a \sqsubseteq_A g(b)$.

That is, every computation sequence included in the first tree is also contained in the second. Under the definition, we can have, for example

$$
\begin{array}{ccc}
\begin{array}{cc}
\bigtriangledown & \bigtriangledown \\
\alpha_1 \vdash \mathit{true} \Downarrow \mathit{true} & \alpha_1 \vdash e_1 \Downarrow a_1 \\
\backslash & / \\
\multicolumn{2}{c}{\alpha_1 \vdash \texttt{if}\ \mathit{true}\ e_1\ e_2 \Downarrow a_1}
\end{array}
& \preceq &
\begin{array}{ccc}
\bigtriangledown & \bigtriangledown & \bigtriangledown \\
\alpha_1 \vdash \mathit{true} \Downarrow \top & \alpha_1 \vdash e_1 \Downarrow a_1 & \alpha_1 \vdash e_2 \Downarrow a_2 \\
\backslash & | & / \\
\multicolumn{3}{c}{\alpha_0 \vdash \texttt{if}\ \mathit{true}\ e_1\ e_2 \Downarrow a_1 \sqcup a_2}
\end{array}
\end{array}
$$

With a bit of effort, we can show that $\mathcal{U}_A$ is a complete lattice. Next, we define the well-formed abstract semantics trees to be $gfp\bar{\Phi}$. As before, for given $\theta \in \mathit{Abstract\_env}$, $e \in \mathit{Expression}$, there can be multiple abstract semantics trees, and we desire the least one. Let $\mathcal{A}\colon \mathit{Expression} \rightarrow \mathit{Abstract\_env} \rightarrow \mathcal{U}_A$ be the "interpreter" that constructs the least abstract semantics tree.

8 Soundness of the Abstract Semantics

Given the above machinery, we must show that abstract semantics trees safely predict concrete ones. Say that a (U-closed, inclusive) relation, $rel_\iota \subseteq \mathit{Base\text{-}constant} \times \mathit{Abstract\text{-}base\text{-}constant}$, has been defined to relate concrete base constants to abstract ones. Next, our objectives are:

- extend rel_ι to closures, giving a relation $rel_{Value} \subseteq \mathit{Value} \times \mathit{Abstract\_value}$
- define a relation, $rel_{Env} \subseteq \mathit{Env} \times \mathit{Abstract\_env}$, relating concrete and abstract environments
- define a relation, $rel_U \subseteq \mathcal{U} \times \mathcal{U}_A$, that relates concrete to abstract trees
- prove the following soundness result for all programs, e:

$$\rho\ rel_{Env}\ \theta \text{ implies } (\mathcal{E}\ e\ \rho)\ rel_{\mathcal{U}}\ (\mathcal{A}\ e\ \theta) \tag{1}$$

The first three items are largely mechanical. Define $rel_{\mathcal{U}} \subseteq \mathcal{U} \times \mathcal{U}_A$ as the largest relation such that $c\ rel_{\mathcal{U}}\ d$ iff *(i)* $c = \Omega$ or *(ii)* $d = \Delta$ or *(iii)* $root(c) = \rho \vdash e \Downarrow v$, $root(d) = \theta \vdash e \Downarrow a$, $\rho\ rel_{Env}\ \theta$, $v\ rel_{Value}\ a$, and and for all subtrees, i, of $root(c)$, $c_i\ rel_{\mathcal{U}}\ d_i$. The relation $rel_{\mathcal{U}}$ states that for every node in the concrete tree, c, there is a related, homologous node in the abstract tree, d. Again, note that an abstract tree might contain more computation paths than the concrete tree to which it is related.

The relation, rel_{Env}, is the expected one: $\rho\ rel_{Env}\ \theta$ if $domain(\rho) = domain(\theta)$ and for all $i \in domain(\rho), \rho(i)\ rel_{Value}\ \theta(i)$

This leaves the definition of rel_{Value} for closures. The relation should be defined in the style of a logical relation: for concrete closure $f \in \mathit{Value}$, abstract closure $g \in \mathit{Abstract\_value}$, $f\ rel_{Value}\ g$ if for all $v \in \mathit{Value}$, $a \in \mathit{Abstract\_value}$, $v\ rel_{Value}\ a$ implies

[9] To make last the clause of the definition of $\preceq$ well formed, some standard indexing of subtrees is required, say, based on *Label*. Or, one might state instead, "and for every subtree, i, of t, there exists a subtree, j, of u, such that $t_i \preceq u_j$."

$apply_C(f, v)\, rel_{\mathcal{U}}\, apply_A(g, a)$, where $apply_C : Value \times Value \rightarrow \mathcal{U}$ evaluates a closure applied to its argument (similarly for $apply_A$).

The mutually recursive definition of rel_D, $D \in \{Value, Env, \mathcal{U}\}$, is resolved by taking the largest relation satisfying the above constraints.

Since the source language in Figure 2 is simply typed, we could index rel by types—if we did we would note that the definition of rel for closures is in fact stronger than a traditional logical relation, because $f\ rel_{Value}\ g$ requires not only that $val(root(apply_C(f, v)))\ rel_{Value}\ val(root(apply_A(g, a)))$ but also that the trees used to create the related results be themselves related. In this way, rel_{Value} is an "intensional" form of logical relation.

If the starting relation, rel_ι, is inclusive and U-closed, fixed-point coinduction lets us prove that rel_D, $D \in \{Value, Env, \mathcal{U}\}$, is inclusive and U-closed as well.

Finally, we are ready to deal with the proof of soundness, (1) above. The proof is performed by an induction on the structure of $e \in Expression$. The induction amounts to an examination of each rule scheme to verify that the concrete semantics version of the rule scheme is related to the abstract semantics version. The proof case for the `fix` rule scheme relies crucially on the inclusiveness of rel.

9 An Example: First-Order Signs Analysis

We apply the above methodology to a quick proof of the soundness of first-order signs analysis for the language in Figure 1. The *Value* and *Env* sets in Figure 2 are replaced by the following:

$$\begin{array}{rcl} a & ::= & neg \ \mid\ 0pos \ \mid\ \langle x, e, \theta\rangle \ \mid\ \bot \ \mid\ \top \\ \theta & ::= & \{i = v_i\}_{i \in Id},\ Id \subseteq Identifier \end{array}$$

The change is that the base constants are replaced by sign properties (including $\top$). The sets are defined coinductively and the following (largest) partial ordering is defined:

- $\bot \sqsubseteq_{Abstract\_value} a$ and $a \sqsubseteq_{Abstract\_value} \top$, for all $a \in Abstract\_value$
- $neg \sqsubseteq_{Abstract\_value} neg$ and $0pos \sqsubseteq_{Abstract\_value} 0pos$
- $\langle x_1, e_1, \theta_1\rangle \sqsubseteq_{Abstract\_Value} \langle x_2, e_2, \theta_2\rangle$ iff $x_1 = x_2, e_1 = e_2$, and $\theta_1 \sqsubseteq_{Abstract\_Env} \theta_2$
- $\theta_1 \sqsubseteq_{Abstract\_Env} \theta_2$ iff $domain(\theta_1) = domain(\theta_2)$ and for all $i \in domain(\theta_1)$, $\theta_1(i) \sqsubseteq_{Abstract\_Value} \theta_2(i)$

Note that the join of two dissimilar closures is $\top$. In this sense, signs analysis is "first order." When the partial ordering is relaxed to allow joins of dissimilar closures, a "closures analysis" is the result, as we will see in Section 12.

The relationship between concrete and abstract base constants goes as follows:

- $\bot\ rel_{Value}\ \bot$
- $n\ rel_{Value}\ 0pos$, if $n \geq 0$; $n\ rel_{Value}\ neg$, if $n < 0$
- $c\ rel_{Value}\ \top$, for all $c \in Value$

Now, we extend this relation to closures and environments in the fashion described in Section 8:

- $\langle x_1, e_1, \rho\rangle \; rel_{Value} \; \langle x_2, e_2, \theta\rangle$ if for all $v \in$ *Value*, $a \in$ *Abstract_value*, $v \; rel_{Value} \; a$ implies $(\mathcal{E}\, e_1\, (\rho \oplus (x_1 = v)))\; rel_{\mathcal{U}}\; (\mathcal{A}\, e_2\, (\theta \oplus (x_2 = a)))$
- $\rho \; rel_{Env}\; \theta$ if $domain(\rho) = domain(\theta)$ and for all $i \in domain(\rho)$, $\rho(i) \; rel_{Value}\; \theta(i)$

Finally, $rel_{\mathcal{U}}$ is defined as stated in Section 8.

The new *Abstract_value* set impacts only the inference rule schemes in Figure 4 for constants, addition, and conditional:

$$\theta \vdash +n \Downarrow 0pos \qquad \frac{\theta \vdash e_1 \Downarrow v_1 \quad \theta \vdash e_2 \Downarrow v_2}{\theta \vdash e_1 \, + \, e_2 \Downarrow absplus(v_1, v_2)} \qquad \frac{\theta \vdash e_2 \Downarrow v_2 \quad \theta \vdash e_3 \Downarrow v_3}{\theta \vdash \mathtt{if}\; e_1\; e_2\; e_3 \Downarrow v_2 \sqcup v_3}$$

$$\theta \vdash -n \Downarrow neg$$

where $absplus(0pos, 0pos) = 0pos$, $absplus(0pos, neg) = \top$, etc. The other rule schemes stay the same. The relation, $\preceq$, between concrete and abstract trees is basically as described in Section 7.

The proof of soundness proceeds as described in the previous section: the rule schemes are examined one by one, but only the schemes for constants, addition, and conditional are of interest—since the rule schemes for the other cases are identical for both concrete and abstract semantics, a variant of the "basic lemma for logical relations" [28] ensures that soundness holds.

Finally, we observe that the only infinite abstract values used in the abstract semantics are the ones generated by the rule scheme for `fix`. Therefore, we can encode the elements of the coinductively defined *Abstract_value* set by the following inductively defined set:

$$a \;\; ::= \;\; \bot \;\; | \;\; neg \;\; | \;\; 0pos \;\; | \;\; \top \;\; | \;\; \langle x, e, \theta\rangle \;\; | \;\; \mu f\langle x, e, \theta\rangle$$

The last construction, a μ-closure, encodes the infinite closure generated by the rule for `fix`: $\mu f\langle x, e, \theta\rangle$ encodes $c_\infty = \langle x, e, \theta \oplus (f = c_\infty)\rangle$. An important consequence of this encoding is, for every $e \in$ *Expression* and $\theta \in$ *Abstract_env*, we are guaranteed that the subset of *Abstract_value* used in the abstract semantics of e, θ is finite.

This last point ensures that the abstract semantics defined in this section can be implemented as a finitely convergent data flow analysis. This topic is examined in the next section.

10 Abstract Collecting Semantics and Finite Convergence

What we call a "data flow analysis" (*d.f.a.*) is just an abstract collecting semantics:

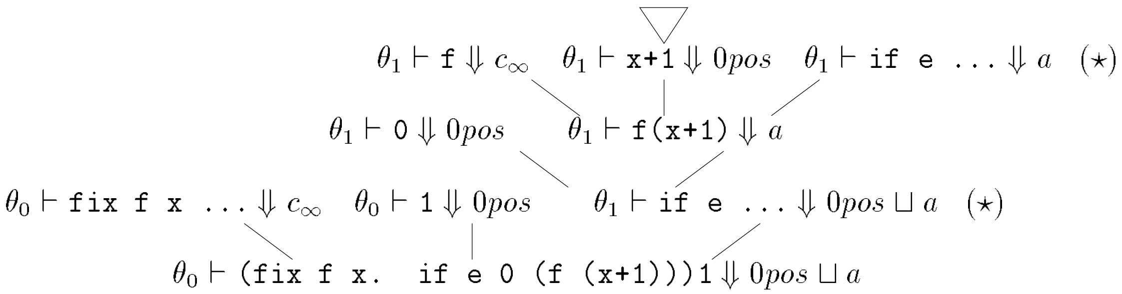
$$coll_{\sqcup}(d) = \lambda l \in Label.\ \bigsqcup\{(\theta, a) \;\; | \;\; \theta \vdash l : e \Downarrow a \text{ is a node of } d\}$$

Here, the join operator, $\bigsqcup$, is typically the join operation for *Abstract_env* $\times$ *Abstract_value*. Thanks to the soundness proof of the earlier section, the abstract collecting semantics safely predicts the concrete one. In data flow analysis terminology, the abstract collecting semantics is called the *meet-over-all-paths solution* [23].

But finite convergence remains an issue: the abstract collecting semantics is defined in terms of the abstract semantics tree, and the latter usually has infinite paths. Since a data flow analysis is expected to terminate, the building of the abstract semantics tree must be bounded but the abstract collecting semantics must be preserved.

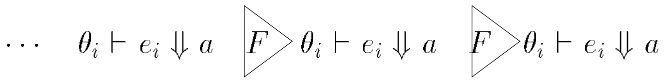
First, consider the situation that, for e_0, θ_0, the sets *Abstract_value* and *Abstract_env* are finite. (In the case of the signs analysis example, this holds.) Immediately we have: for every path in the tree $\mathcal{A}\, e_0\, \theta_0$, either the path is finite (it terminates), or there is a *repetition node*, $\theta_i \vdash e_i \Downarrow a$, in the path (that is, the node appears more than once in the path). The key to detecting convergence is to use the abstract interpreter, $\mathcal{A}$, to unfold the abstract semantics tree until every path terminates or contains a repetition node. Then, one uses least-fixed point techniques to solve for the value, a, in the repetition node.

An example will give good intuition. Say that signs analysis is applied to the program `(fix f x.  if e 0 (f(x+1)))1` with an initial abstract environment, θ_0. When we unfold the abstract semantics tree, we see the following. (Let $c_\infty = \mu f \langle$`x, if e 0 (f(x+1))`$, \theta_0\rangle$ and let $\theta_1 = \theta_0 \oplus (\mathtt{f} = c_\infty) \oplus (\mathtt{x} = 0pos)$.)



At the nodes marked $(\star)$, there is a repetition node—the sequent $\theta_1 \vdash$ `if e ...` $\Downarrow$ ... has repeated. The unfolding of the upper right node in the tree will repeat what is already present, and it must be the case that $a = 0pos \sqcup a$, anyway. So, rather than unfold the tree further, we can solve for a and quit.

Here is the formal justification. Say that we used $\mathcal{A}\; e_i\; \theta_i$ to generate a tree, t, all of whose paths but one are finite; the remaining path contains a repetition of the root node, $\theta_i \vdash e_i \Downarrow v$. The situation is best seen in a diagram. (The tree has been rotated 90 degrees to save space; the root is rightmost.)



Say that F is the tree scheme that generates the repetition. We have the following equalities:

$$t = \mathcal{A}\, e_i\, \theta_i = F(\mathcal{A}\, e_i\, \theta_i) = F(t) = \mathit{lfp} F = \bigsqcup_{i \geq 0} t_i, \text{ where } t_0 = \Omega \text{ and } t_{i+1} = F(t_i)$$

Next, for a tree, t', whose topmost part is generated by a tree scheme, F, define $\mathit{top}_F(t')$ to be that topmost part of t' that matches F. By induction on i, we have, for all $i > 0, \mathit{coll}_\sqcup(t_i) = \mathit{coll}_\sqcup(\mathit{top}_F(t_i))$. An immediate corollary is that $\mathit{coll}_\sqcup(t) = \mathit{coll}_\sqcup(\mathit{top}_F(t))$, that is, the collecting semantics of t can be defined from just the top F-part of it.

This proposition gives us the technique for detecting convergence: for a tree $t = F(t)$, we obtain the correct collecting semantics from $t' = \bigsqcup_{i \geq 0} t'_i$, where $t'_0 = \Omega$, and $t'_{i+1} = F(\theta_i \vdash e_i \Downarrow val(root(t'_i)))$ where $val(\theta \vdash e \Downarrow a) = a$ and $val(\Omega) = \bot$. That is, $\theta_i \vdash e_i \Downarrow val(root(t'_i))$ is a leaf and *not* the entire tree—the abbreviated representation, t', does not generate the entire tree, t; instead, it stops at the first repetition of $\theta_i \vdash e_i \Downarrow a$ and solves for the value, a, that appears at the repetition node. In the universe of abstract trees, $\mathcal{U}_A$, a least upper bound of the t'_is exists, and the finiteness of *Value* ensures that $t'_j = \bigsqcup_{i \geq 0} t'_i$, for some j. Since F effectively defines a system of first-order data-flow equations for the abstract environments and values within it, we have that $t'_{j+1} = t'_j$ implies $t'_j = \bigsqcup_{i \geq 0} t'_i$. Thanks to the earlier proposition in this section, we have $coll_{\sqcup}(t'_j) = coll_{\sqcup}(t)$—the meet-over-all-paths solution is preserved.

11 Convergence with Infinite Abstract Domains

When *Abstract_value* is an infinite set, there is no guarantee that every path in an abstract semantics tree terminates or has a repetition node. If the ordering on *Abstract_value* has finite height, however, one can use the classical technique from data flow analysis to force every path to have a repetition. From the data flow analysis perspective, a program is a flowchart, and the input abstract environment to a phrase, $l\colon e$, is defined by the flow equation $\theta_l = \bigsqcup_{m \in pred(l)} out\_value(m)$, where $out\_value(m) = [\![m\colon e']\!]\theta_m$.

An example shows best how the flow equation definition of θ_l is applied to an abstract semantics tree. Once again, we consider the example program `(fix f x. if e 0 (f(x+1)))1` and input environment θ_0, but this time we perform a constant propogation analysis. This differs from a signs analysis in that the abstract base constants are the integers; we use the same inference rule schemes as for signs analysis, but the *absplus* operation used for $e_1 + e_2$ changes: $absplus(1,2) = 3$, $absplus(1,\top) = absplus(\top,1) = \top$, etc.

When we unfold the abstract semantics tree, we find the following. (Let $c_\infty = \mu f\langle \texttt{x, if e 0 (f(x+1))}, \theta_0\rangle$; $\theta_1 = \theta_0 \oplus (\texttt{f} = c_\infty) \oplus (\texttt{x} = 1)$; and $\theta_2 = \theta_0 \oplus (\texttt{f} = c_\infty) \oplus (\texttt{x} = 2)$.)

$$
\begin{array}{c}
\qquad\qquad\qquad\qquad\qquad\qquad \triangledown \qquad\qquad\qquad\qquad\qquad \vdots \\
\theta_1 \vdash \texttt{f} \Downarrow c_\infty \quad \theta_1 \vdash \texttt{x+1} \Downarrow 2 \quad \theta_2 \vdash \texttt{if e ...} \Downarrow a \quad (\star\star) \\
\theta_1 \vdash \texttt{0} \Downarrow 0 \qquad \theta_1 \vdash \texttt{f(x+1)} \Downarrow a \\
\theta_0 \vdash \texttt{fix f x ...} \Downarrow c_\infty \quad \theta_0 \vdash \texttt{1} \Downarrow 1 \qquad \theta_1 \vdash \texttt{if e ...} \Downarrow 0 \sqcup a \\
\theta_0 \vdash \texttt{(fix f x. if e 0 (f(x+1)))1} \Downarrow 0 \sqcup a
\end{array}
$$

Notice that there are no repetition nodes along the infinite path, which is rightmost in the tree; the node marked $(\star\star)$ should be the place where repetition arises, but environment θ_2 appears instead of θ_1.

A brute-force way to force a repetition along the infinite path is to alter node $(\star\star)$: since the syntax phrase, `if e ...` has appeared earlier in the tree with environment θ_1, we join θ_1 with the current environment, θ_2. This gives us:

$$\dfrac{\theta_0 \vdash \texttt{fix f x ...} \Downarrow c_\infty \quad \theta_0 \vdash \texttt{1} \Downarrow 1 \quad \dfrac{\theta_1 \vdash \texttt{0} \Downarrow 0 \quad \dfrac{\theta_1 \vdash \texttt{f} \Downarrow c_\infty \quad \dfrac{\bigtriangledown}{\theta_1 \vdash \texttt{x+1} \Downarrow 2} \quad \theta_1 \sqcup \theta_2 \vdash \texttt{if e ...} \Downarrow a \quad (\star\star)}{\theta_1 \vdash \texttt{f(x+1)} \Downarrow a}}{\theta_1 \vdash \texttt{if e ...} \Downarrow 0 \sqcup a}}{\theta_0 \vdash \texttt{(fix f x. if e 0 (f(x+1)))1} \Downarrow 0 \sqcup a}$$

This is *exactly* what happens in a classic, iterative data-flow analysis: the previous input information to a node, l, is joined with the current input to l, forcing an ascending chain of information to develop. Since the abstract value sets are partial orderings of finite height, all paths in the abstract tree must terminate or contain repetition nodes (that is, converge).

Note that the above tree is *not* well formed according to the inference rule schemes, that is, it is not an element of $gfp\bar{\Phi}$. (But it is an element of $\mathcal{U}_A$.) Perhaps the difficulty in formally proving correctness of a classic iterative data-flow analysis to a standard semantics is due to this "abuse" of well formedness. Here, we will obtain an easy safety proof, however, because our safety relation, $rel_{\mathcal{U}}$, is U-closed.

Here is a more formal explanation of the above example. First, for e_0, θ_0, recall that $\mathcal{A}\, e_0\, \theta_0 = \bigsqcup_{i \geq 0} t_i$, where $t_i = \mathcal{A}_i\, e_0\, \theta_0$, where $\mathcal{A}_i$ is the i-unfolding of $\mathcal{A}$. Straightforward but tedious reasoning shows that t_{i+1} is derived from t_i by expanding each of t_i's *frontier nodes*[10] by one unfolding of $\mathcal{A}$.

Now, we alter the calculation of the t_is to proceed as follows: $t''_0 = \mathcal{A}_0\, e_0\, \theta_0 = \Omega$, $t''_1 = \mathcal{A}_1\, e_0\, \theta_0$, and $t''_{i+2} =$ *(i)* alter each frontier node, $\theta_i \vdash l{:}\, e \Downarrow \bot$, in t''_{i+1} to read $\theta'_i \vdash l{:}\, e \Downarrow \bot$, where $\theta'_i = \bigsqcup\{\theta \ \mid\ \theta \vdash l{:}\, e \Downarrow a$ is a node in $t''_{i+1}\}$; and *(ii)* expand each frontier node by one additional unfolding of $\mathcal{A}$. That is, the abstract environment, θ_l, at a frontier node, $\theta_l \vdash l{:}\, e \Downarrow \bot$, is joined with the global information of all other abstract environments that previously flowed into e_l, giving $\theta'_i \vdash l{:}\, e \Downarrow \bot$, which is then unfolded.

This procedure ensures that every path in the abstract semantics tree is finite or contains a repetition node, so the technique from the previous section applies. By monotonicity of $\mathcal{A}$, we have $\mathcal{A}\, e_0\, \theta_0 \preceq t''$, and by the U-closure of $rel_{\mathcal{U}}$, $(\mathcal{E}\ e_0\ \rho_0)\ rel_{\mathcal{U}}\ t''$ holds.

Unfortunately, the collecting semantics for t'' may be larger than the meet-over-all-paths solution. But when the program analyzed is tail-recursive, that is, it has the form, $(\texttt{fix}\ f\ a.\texttt{if}\ e\ \ (g\ a)\ (f\ (h\ a)))a_0$, and the semantics of g and h are distributive, the above method preserves the meet-over-all-paths solution.

12 An Example: Closures Analysis

The methodology developed so far will reveal to us that closures analysis is merely a higher-order abstract interpretation: an abstract closure is generalized from one closure to a finite set of them. Recall that each subphrase of a source program, e, is labelled by a unique $l \in Label$. In particular, this means each lambda-abstraction is uniquely labelled, so we write $l{:}\, \lambda x_l.e_l$ to identify the components, x_l and e_l, of the abstraction labelled by l.

[10] Here, a *frontier node* of a tree is a nonleaf node of the form $\theta \vdash e \Downarrow \bot$ whose subtrees are all Ω.

The sets *Abstract_value* and *Abstract_env* are defined as follows:

$$\begin{array}{lcl} a & ::= & b \ \mid\ \bot \ \mid\ \top \ \mid\ \{l = \theta_l\}_{l \in L}, \ \ L \subseteq_{fin} Label \\ \theta & ::= & \{i = a_i\}_{i \in Id}, \ \ Id \subseteq_{fin} Identifier \end{array}$$

An abstract closure is a set of closures, indexed by labels. (The components, x_l and e_l, of a closure are represented by label l.) For this reason, we will treat an abstract closure as a function of form $Label \rightarrow Abstract\_env$. The base constants, b, can be whatever is desired (e.g., signs).

The partial ordering on *Abstract_value* is modified for the new version of closures; let S_1, S_2 be abstract closures:

$$S_1 \sqsubseteq_{Abstract\_value} S_2 \text{ iff for all } l \in domain(S_1),\ S_1(l) \sqsubseteq_{Abstract\_env} S_2(l)$$

This ordering makes $S_1 \sqcup S_2 = \lambda l \in (domain(S_1) \cup domain(S_2)).\ S_1(l) \sqcup S_2(l)$.

The change in closure representation necessitates these changes in the rule schemes for the abstract semantics of lambda-abstraction and application:

$$\theta \vdash l\colon \lambda x_l.e_l \Downarrow \{l = \theta\}$$

$$\frac{\theta \vdash e_1 \Downarrow \{l = \theta_l\}_{l \in L} \qquad \theta \vdash e_2 \Downarrow a' \qquad \{\theta_l \oplus \{x_l = a'\} \vdash e_l \Downarrow a_l\}_{l \in L}}{\rho \vdash e_1\ e_2 \Downarrow \bigsqcup_{l \in L} a_l} \quad L \subseteq Label$$

When an application, $(e_1 e_2)$, is evaluated, the operator part, e_1, evaluates to a set of closures; each closure in the set is executed with a', which is the value of e_2, and the results are joined to obtain an overall result. A consequence of this strategy is that a program's abstract semantics tree will have more computation paths than the program's concrete semantics tree. But, as was noted in Section 8 in the definition of $rel_{\mathcal{U}}$, this is perfectly acceptable for a sound abstract interpretation.

The proof of soundness of the closures analysis is straightforward: the definitions of *rel* follow the pattern outlined in Section 8; the only novelty is the change of rel_{Value} for closures:

$$\begin{array}{l} \langle x_m, e_m, \rho\rangle\ rel_{Value}\ \{l = \theta_l\}_{l \in L} \text{ iff } m \in L \text{ and for all } v \in Value, a \in Abstract\_value, \\ \quad v\ rel_{Value}\ a \text{ implies } (\mathcal{E}\ e_m\ (\rho \oplus (x_m = v)))\ rel_{\mathcal{U}}\ (\mathcal{A}\ e_m\ (\theta \oplus (x_m = a))) \end{array}$$

Because *rel* is U-closed, the soundness proof is straightforward.

The above modelling of abstract closures is an *independent attribute* analysis[11] [19]—when two closures, $\{l_0 = \theta\}$ and $\{l_0 = \theta'\}$ are joined, the result is $\{l_0 = \theta \sqcup \theta'\}$ instead of $\{l_0 = \{\theta, \theta'\}\}$. If the latter is desired, the analysis becomes a *relational analysis.*[12]

To define a relational closures analysis, the representation of abstract closures is changed to the form $Label \rightarrow \wp(Abstract\_env)$, and for abstract closures, S_1 and S_2,

$$S_1 \sqsubseteq_{Abstract_value} S_2 \text{ iff for all } l \in domain(S_1),\ S_1(l) \subseteq S_2(l)$$

As noted earlier, the *d.f.a.* that results from the above formulation is the collecting semantics of the abstract semantics tree. And there is also the issue of finite convergence:

[11]That is, an analysis based on a *computational ordering* [8].

[12]That is, an analysis based on an *approximation ordering.*

when the source language is simply-typed, as is the case in Figure 1, and the set of abstract base constants are finite, it appears to be the case that the *Abstract_value* set is finite as well.[13] This ensures convergence of the closures analysis.

The usual information desired from closures analysis is the set of closures that a labelled phrase, l, might evaluate to. Given our definition of a *d.f.a.* , namely, the collecting semantics of the abstract semantics tree, t, the information we desire is $coll_{\sqcup}(t)(l) \downarrow 2$.

12.1 Relation to Existing Implementions of Closures Analysis

Sestoft [39] coined the term "closures analysis" for his algorithm that calculated the sets of closures that phrases in a program evaluate to. Sestoft's algorithm is coded in a compositional, denotational-semantics style, and uses two data structures: a global, abstract environment of form $Label \rightarrow Abstract\_value$ (recall that binding identifiers are labelled) and a table that maps the bodies of lambda abstractions to the closures they evaluate to. The latter is not essential to the algorithm, but it simplifies the presentation.

In Sestoft's algorithm, an abstract closure is a set of labels—no environments are saved in closures. When an abstract closure is invoked, the global abstract environment is used with each of the labels in the abstract closure. A practical consequence of this strategy is that the arguments (environments) of all invocations of a closure, l, are joined together in the global abstract environment—the global environment does double duty as a cache.

In the terminology of the previous section, Sestoft's closures analysis algorithm is an independent-attribute closures analysis where the frontier nodes of the abstract tree are updated with global information as described in Section 11. Palsberg [37] gives a terse reformulation of Sestoft's algorithm.

In [41, 42], Shivers describes several control-flow-analysis algorithms and ranks them by precision. The algorithms are coded as denotational semantics-style definitions and use a cache to collect closures information. Environments are modelled as displays: $Benv = Label \rightarrow Contour$, where $Label$ labels a lambda-abstraction, and $Contour$ is a pointer to an activation record in a global heap where the binding(s) for a lambda abstraction's binding identifiers are stored. A concrete closure is an element of form $Label \times Benv$.

Shivers's "exact flow control semantics" is a coding of a collecting semantics within this framework. His 0-CFA abstract semantics defines abstract closures to have form $\wp(Label \times Contour_0)$, where $Contour_0 = \{\bullet\}$. That is, each lambda abstraction is allowed at most one activation record in the global heap. This means that the arguments (environments) of all invocations of a lambda abstraction, l, are joined together in l's sole activation record in the heap. The 0-CFA algorithm calculates the same information as Sestoft's algorithm.

Shivers notes that a more precise analysis, 1-CFA, can be obtained by expanding the contour set for each lambda abstraction to be the set of "call sites" in the program. (A call site is the label of the operator part an application expression.) Then, only those uses of the same lambda abstraction, l, at the same call site, c, have their arguments (environments) joined together. A 1-CFA analysis is a simple-minded relational closures analysis, where

[13] The informal justification goes as follows: since the *Label* set is finite, closures of the form $\{l = \{...l = \{...\}...\}\}$ are necessary for an infinite *Abstract_value* set. But such closures can be built only from self-application, which is impossible in the simply-typed lambda calculus. (Sorry—I didn't have time to write this proof properly before the submission deadline!)

each frontier node in the abstract semantics tree that is a closure at a call site is updated by the global information previously encountered by that closure at that same call site.

The notion of n-CFA, for $n > 1$, can be defined by "call strings," which are n-sequences of labels that list the expressions that evaluate to a closure, c, prior to the execution of c. Stefanescu and Zhou [43] use equivalence relations on call strings to define a hierachy of n-CFA analyses. In terms of the developments in this paper, their equivalence relations determine when frontier nodes in the abstract semantics tree are updated with global information.

Jagannathan and Weeks [17] show how Shivers's and Stefanescu and Zhou's analyses can be described by control flow graphs derived from a state-transition operational semantics.

Finally, Jagannathan and Wright [18] point out that a good-precision analysis can be had by performing a "polymorphic splitting" of a program and then proceeding with a 0-CFA analysis. "Splitting" means that a program,

$$\texttt{let f=}\ l_0\texttt{:}\ \lambda\texttt{x.e in}\ \ldots\ l_1\texttt{:f}\ \ldots\ l_i\texttt{:f}\ \ldots$$

is rewritten into

$$\texttt{let f}_1\texttt{=}\ l_{01}\texttt{:}\ \lambda\texttt{x.e}\ \ldots\ \texttt{let f}_i\texttt{=}\ l_{0i}\texttt{:}\ \lambda\texttt{x.e}\ \ldots\ \texttt{in}\ l_1\texttt{:f}_1\ \ldots\ l_i\texttt{:f}_i\ \ldots$$

In practice, this analysis often does better than 1-CFA.

13 Conclusion

We have demonstrated that natural semantics is a useful vehicle for stating the basic steps of an abstract interpretation. Specification issues are separated from implementation issues, allowing straightforward correctness proofs. In the case of closures analysis, we saw that the existing closures analysis algorithms are standard forms of independent-attribute analysis and relational analysis, modified for efficiency and convergence.

Although this paper has emphasized the reformulation of classical abstract intepretation in natural semantics, recent work by Gouranton and LeMétayer [12] show how modern analysis algorithms can be derived from parameterized properties extracted from paths in natural semantics trees.

14 Acknowledgement

On short notice, Daniel LeMétayer read an earlier draft of this paper and made many helpful suggestions.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[2] A. Banerjee. *The Semantics and Implementation of Bindings in Higher-Order Programming Languages.* PhD thesis, Kansas State University, 1995.

[3] A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.

[4] G. L. Burn, C. Hankin, and S. Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. In *Proc. 4th ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[6] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. 6th ACM Symp. on Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

[7] P. Cousot and R. Cousot. Inductive definitions, semantics, and abstract interpretation. In *Proc. 19th ACM Symp. on Principles of Programming Languages*, pages 83–94. ACM Press, 1992.

[8] P. Cousot and R. Cousot. Higher-order abstract interpretation. In *Proc. IEEE Int'l. Conf. Programming Languages*. IEEE Press, 1994.

[9] Th. Despeyroux. Executable specification of static semantics. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 215–234. Lecture Notes in Computer Science 173, Springer-Verlag, 1984.

[10] V. Donzeau-Gouge. Denotational definition of properties of program's computations. In S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[11] J. Goguen, J. Thatcher, E. Wagner, and J. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24:68–95, 1977.

[12] V. Gouranton and D. LeMétayer. Derivation of static analysers of functional programs from path properties of a natural semantics. Technical Report Research Report 2607, INRIA, 1995.

[13] I. Guessarian. *Algebraic Semantics*. Springer Lecture Notes in Computer Science 99. Springer-Verlag, 1981.

[14] C. Gunter. *Foundations of Programming Languages*. MIT Press, Cambridge, MA, 1992.

[15] M. Hecht. *Flow Analysis of Computer Programs*. Elsevier, 1977.

[16] P. Hudak and J. Young. A collecting interpretation of expressions (without powerdomains). In *Proc. 15th ACM Symp. on Principles of Programming Languages*, pages 107–118. ACM Press, 1988.

[17] S. Jagannathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In *Proc. 22d. ACM Symp. Principles of Programming Languages*, pages 393–407, 1995.

[18] S. Jagannathan and A. Wright. Effective flow analysis for avoiding run-time checks. In *Proc. Workshop on types for program analysis*, number `http://www.daimi.aau.dk/~bra8130/TPA/proceedings.html`, Univ. of Aarhus, Denmark, 1995.

[19] N.D. Jones and S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Proc. 6th. ACM Symp. Principles of Programming Languages*, pages 244–256, 1979.

[20] N.D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Proc. 13th Symp. on Principles of Prog. Languages*, pages 296–306. ACM Press, 1986.

[21] N.D. Jones and M. Rosendahl. Higher-order minimal function graphs. In *???* ???, 1994.

[22] G. Kahn. Natural semantics. In *Proc. STACS '87*, pages 22–39. Lecture Notes in Computer Science 247, Springer, Berlin, 1987.

[23] J. Kam and J. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23:158–171, 1976.

[24] G. Kildall. A unified approach to global program optimization. In *Proc. ACM Symp. on Principles of Programming Languages*, 1973.

[25] P. Lewis, D. Rosenkrantz, and R. Stearns. *Compiler Design Theory.* Addison-Wesley, 1978.

[26] A. Melton, G. Strecker, and D. Schmidt. Galois connections and computer science applications. In *Category Theory and Computer Programming*, pages 299–312. Lecture Notes in Computer Science 240, Springer-Verlag, 1985.

[27] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 17:209–220, 1992.

[28] J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 8, pages 367–458. Elsevier Science Publishers, 1990.

[29] M. Mizuno and D. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4:727–754, 1992.

[30] S. Muchnick and N.D. Jones, editors. *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.

[31] A. Mycroft. *Abstract interpretation and optimizing transformations for recursive programs.* PhD thesis, Edinburgh University, 1981.

[32] A. Mycroft and N.D. Jones. A relational framework for abstract interpretation. In *Programs as Data Objects*, pages 156–171. Lecture Notes in Computer Science 217, Springer-Verlag, 1985.

[33] F. Nielson. Semantic foundations of data flow analysis. Technical Report Report DAIMI PB-131, Aarhus University, Denmark, 1981.

[34] F. Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

[35] F. Nielson. Program transformations in a denotational setting. *ACM Trans. Prog. Languages and Systems*, 7:359–379, 1985.

[36] F. Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, 69(2):117–242, 1989.

[37] J. Palsberg. Global program analysis in constraint form. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Proc. CAAP'94*, Lecture Notes in Computer Science, pages 258–269. Springer-Verlag, 1994.

[38] G. D. Plotkin. Lambda-definability in the full type hierarchy. In J. Seldin and J. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363–374. Academic Press, 1980.

[39] P. Sestoft. Replacing function parameters by global variables. In *Proc. Functional Programming and Computer Architecture*, pages 39–53. ACM Press, 1989.

[40] P. Sestoft. *Analysis and Efficient Implementation of Functional Programs.* PhD thesis, Copenhagen University, 1991.

[41] O. Shivers. Control-flow analysis in Scheme. In *Proc. SIGPLAN88 Conf. on Prog. Language Design and Implementation*, pages 164–174, 1988.

[42] O. Shivers. *Control Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, 1991.

[43] D. Stefanescu and Y. Zhou. An equational framework for flow analysis of higher-order functional programs. In *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, pages 318–327. ACM Press, 1994.