Trace-Based Abstract Interpretation of Operational Semantics

David A. Schmidt
Kansas State University*

Abstract

We present *trace-based abstract interpretation*, a unification of several lines of research on applying Cousot-Cousot-style abstract interpretation (*a.i.*) to operational semantics definitions (such as flowchart, big-step, and small-step semantics) that express a program's semantics as a concrete computation tree of trace paths. A program's traced-based *a.i.* is also a computation tree whose nodes contain abstractions of state and whose paths simulate the paths in the program's concrete computation tree. Using such computation trees, we provide a simple explanation of the central concept of *collecting semantics*, and we distinguish concrete from abstract collecting semantics and state-based from path-based collecting semantics. We also expose the relationship between collecting semantics extraction and results garnered from flow-analytic and model-checking-based analysis techniques. We adapt concepts from concurrency theory to formalize "safe" and "live" *a.i.* s for computation trees; in particular, coinduction techniques help extend fundamental results to infinite computation trees.

Problems specific to the various operational semantics methodologies are discussed: Big-step semantics cannot express divergence, so we employ a mixture of induction and coinduction in response; small-step semantics generate sequences of program configurations unbounded in size, so we abstractly interpret source language syntax. Applications of trace-based *a.i.* to data-flow analysis, model checking, closure analysis, and concurrency theory are demonstrated.

1 Introduction

Abstract interpretation (*a.i.*) is accepted as the correctness foundation for data-flow analysis of flowchart programs [17, 18, 41], and related research has demonstrated that *a.i.* can be applied to nonflowchart programs defined by denotational semantics [2, 8, 21, 25, 41, 45, 55, 63, 57, 58, 59] and structural operational semantics [19, 30, 68, 69, 70, 71, 81]. Model checking is another important applications area [6, 10, 22, 23, 76, 77].

The theory of *a.i.* of denotational semantics definitions is mature, and we present an initial unification of several lines of research on the *a.i.* of operational semantics definitions. Common to all of flowchart (state-transition), big-step (natural), and small-step (SOS) operational semantics is the notion of *computation tree*: A program's concrete semantics is a tree whose paths represent execution traces and whose nodes display the program's changing states. Therefore, the program's *a.i.* must also be a computation tree whose nodes contain abstractions of state and whose paths simulate the paths in the corresponding concrete

*Computing and Information Sciences Department, 234 Nichols Hall, Manhattan, KS 66506 USA. `schmidt@cis.ksu.edu`. Supported by NSF CCR-9302962 and CCR-9633388.

computation tree. For this reason, we coin the term *trace-based abstract interpretation* to denote techniques for *a.i.* of operational semantics that build computation trees of traces.

Perhaps the primary objective of an *a.i.* is to calculate a program's *collecting semantics*, and a major contribution of this paper is a simple definition of collecting semantics extraction from a computation tree. Also, we clarify the distinctions between concrete and abstract collecting semantics, between state-based and path-based collecting semantics, and we expose the relationship between collecting semantics extraction and flow-analytic-based and model-checking-based static analysis techniques.

We adapt concepts from concurrency theory to formalize "safe" and "live" *a.i.* s for computation trees; in particular, coinduction techniques help extend fundamental results to infinite computation trees.

In addition, we address challenges that arise within the specific semantics forms: Big-step semantics cannot express divergence, so we employ a mixture of induction and coinduction; small-step semantics generate sequences of program configurations unbounded in size, so we abstractly interpret source language syntax in response. Applications of trace-based *a.i.* to data-flow analysis, model checking, closure analysis, and concurrency theory are demonstrated.

Many of the paper's technical concepts are taken from the trailblazing research of Cousot and Cousot [16, 17, 18, 19, 20, 21]—indeed, the first mention of *a.i.* of execution traces appears in P. Cousot's doctoral thesis [16]. Our primary contribution is an expository one: We unify several lines of research in abstract interpretation and concurrency theory to achieve a simple methodology for *a.i.* upon operational semantics.

The structure of the paper goes as follows: Basic concepts appear in Section 1.1; Section 2 applies the concepts to a thorough redevelopment of abstract interpretation of flowchart semantics. Section 3 presents key aspects of definition and proof by coinduction. Section 4 surveys briefly liveness abstract interpretations, and Sections 5 and 6 present traced-based *a.i.* for big-step semantics and small-step semantics, respectively, reaffirming the methodology's utility and addressing problems specific to these formats. Applications are intertwined with the semantic forms upon which they are based. Section 7 concludes.

1.1 What is Trace-Based Abstract Interpretation?

An operational semantics that defines run-time executions is called a *concrete semantics* or *concrete interpretation* (*c.i.*). Using the concrete semantics, a program plus its run-time input data can be executed; the result is a *concrete computation tree*, or *concrete tree*, for short. The tree's paths form an "execution trace," which displays at the tree's nodes the changing state of the computation; if the computation is divergent, an infinite tree results.

If the run-time data is abstracted to a set of "tokens" that represent properties of the data and if the semantics is revised to compute upon such tokens, one obtains an *abstract semantics* or *abstract interpretation* (*a.i.*). A program plus an input data token can be interpreted by the abstract semantics; the result is again a tree, called the *abstract computation tree*, or *abstract tree*, for short.

It is convenient to think of an *a.i.* as a "symbolic execution" where the symbols have semantic content. One example is the implementation of type inference by an *a.i.* where run-time data are replaced by datatype tokens—e.g., data like *2* and *true* are replaced by *int* and *bool*, respectively—and the program executes on datatype tokens.

Nondeterminism in the abstract or concrete semantics will derive computation trees that display multiple execution paths and perhaps lead to multiple outcomes. Nondeterminism is the norm in *a.i.* , where a loss of precision in the token set makes many language operators nondeterministic in their outcomes.

When run-time data sets are replaced by tokens, the operators/rules in the semantics must be revised to compute consistently on the tokens, e.g., a rule that defines addition on concrete integers must be revised to define "addition" on datatype tokens. In algebraic terminology, a language's operators define a "signature"; when the operators/rules are instantiated by operations that compute on run-time data, one obtains a *c.i.* of the signature; when the operators/rules are instantiated by operations on tokens, one obtains an *a.i.* of the signature; and when there is a homomorphism from the *c.i.* into the *a.i.* , then the *a.i.* is a *safe simulation* of the *c.i.* (There also exist "live simulations," which are discussed later.) For example, the concrete semantics of `y:=x+1` is the usual assignment operation on `y`, whereas a type-inference abstract semantics goes: `y` is assigned t, if `x`'s value is $t \in \{int, real\}$, else `y` is assigned $\top$ (error type).

A crucial issue is termination: Although the *c.i.* of a program with its run-time data might terminate, the *a.i.* might not, because the tokens are less precise and nondeterminism arises. For example, the abstract interpretation of a test, `x>0`, cannot be decided when the token value of `x` is *int*. This forces the *a.i.* to traverse both execution paths that emanate from the test, implying that loops can be traversed forever. Therefore, an *a.i.* must be coupled with a strategy for termination. One strategy builds a program's abstract tree such that every infinite path in the tree contains a node that is a repetition of one seen earlier in the path, that is, the trace is a *regular tree* [15]—trace building can be terminated at these "repetition nodes." This strategy is employed in this paper.

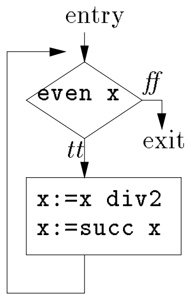
Once a computation tree, whether concrete or abstract, is built, one must extract information from it and apply the information to validation or code improvement. The information extracted is the *collecting semantics*. As just implied, both a concrete tree and an abstract tree possess collecting semantics, called respectively the *concrete collecting semantics* and the *abstract collecting semantics*. A collecting semantics can be state-based or path-based: A state-based ("first-order" [57]) collecting semantics is a mapping from a program's program points (flowchart boxes) to the input domains of the program points. That is, the collecting semantics defines the range of values that enter a program point. A path-based ("second-order") collecting semantics maps a program point to the set of execution paths that lead into (or, dually, lead out from) the program point. An *a.i.* that is a safe simulation of a *c.i.* will produce a collecting semantics that is a superset of the homomorphic image of the one for the *c.i.*

For example, the usual collecting semantics for a type inference is state based, whereas the collecting semantics for an available-expressions analysis is (forwards) path-based, and a live-variable analysis produces a (backwards) path-based collecting semantics. There exist more general forms of collecting semantics [19], which are discussed later.

For efficiency, an implementation of an *a.i.* will build a compact representation of an abstract tree or even bypass the tree and construct a representation of its collecting semantics directly: The cache computed by a flow analysis is a classical example [3], and the cache computed by denotational-semantics analysis is another [36]. We examine the relationship between collecting semantics extraction and these implementations later in the paper.

In summary, the methodology we propose for trace-based *a.i.* proceeds as follows:

1. One begins with a concrete semantics (*c.i.*) that uses a set of run-time data and



Concrete Semantics:
$Val = Nat$
$(2n \vdash \texttt{even x}) \rightarrow (2n \vdash \texttt{x := x div2})$
$(2n+1 \vdash \texttt{even x}) \rightarrow (2n+1 \vdash exit)$
$(2n \vdash \texttt{x := x div2}) \rightarrow (n \vdash \texttt{x := succ x})$
$(n \vdash \texttt{x := succ x}) \rightarrow (n+1 \vdash \texttt{even x})$

Concrete tree:
$4 \vdash \texttt{even x}$
$\downarrow$
$4 \vdash \texttt{x := x div2}$
$\downarrow$
$2 \vdash \texttt{x := succ x}$
$\downarrow$
$3 \vdash \texttt{even x}$
$\downarrow$
$3 \vdash$ exit

Figure 1: Flowchart and concrete interpretation

rules/operators to generate concrete computation trees for programs and their inputs.

2. The run-time data set is abstracted to a set of properties, and the rules/operators are revised to compute upon the properties. The resulting abstract semantics (*a.i.*) generates abstract computation trees.

3. A homomorphism property is used to prove that the *a.i.* is a safe simulation of the *c.i.*

4. If necessary, a strategy is formulated for ensuring termination of abstract tree construction.

5. Collecting semantics is extracted from the abstract trees.

We begin by developing these steps for flowchart semantics.

2 Abstract Interpretation of Flowchart Programs

The principles of abstract interpretation were established for flowchart programs by Cousot and Cousot [17], and the material in this section is a restatement of their work in the trace-based methodology. Precedents for use of computation trees are found in [16, 20, 42, 41], and the associated simulation and coinduction notions stem from [22, 53, 54, 66].

Figure 1 shows a flowchart program that uses a storage vector with a single variable, `x`. A state is a storage vector, program point pair, $v \vdash pp$, where v is the value of `x` and pp is the current program point,[1] and the concrete semantics rules specific to the flowchart are listed in the middle column of the Figure. When executed with input 4, the program's concrete tree has one path. The trace in the Figure is finite, but a divergent program would generate an infinite trace.

Perhaps better target code can be generated for commands whose inputs are always even numbers. This motivates an abstract semantics of the form displayed in Figure 2. The *Val* set is abstracted to $AbsVal = \{e, o\}$, denoting even and odd numbers, respectively, and each concrete transition rule ("operator") is revised into one or more abstract rules.

[1] In the general case, program points are a unique numbering of the statements in a program, but for simplicity, we equate statements with program points.

Abstract Transitions:

$AbsVal = \{e, o\}$

$(e \vdash \texttt{even x}) \to (e \vdash \texttt{x := x div2})$
$(o \vdash \texttt{even x}) \to (o \vdash exit)$
$(e \vdash \texttt{x := x div2}) \to (e \vdash \texttt{x := succ x})$
$(e \vdash \texttt{x := x div2}) \to (o \vdash \texttt{x := succ x})$,
$(e \vdash \texttt{x := succ x}) \to (o \vdash \texttt{even x})$
$(o \vdash \texttt{x := succ x}) \to (e \vdash \texttt{even x})$

Abstract tree:

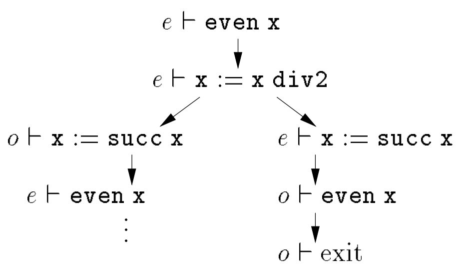


Figure 2: Abstract interpretation of flowchart

For reasons related to precision and termination, one might partially order the elements of *AbsVal* and demand that the abstract transition rules behave monotonically. For this simple example, the discrete ordering on *AbsVal* suffices. The resulting abstract semantics is nondeterministic in its interpretation of `div2`, implying that the abstract tree is nondeterministic.

The abstract tree contains more paths than what appear in the concrete tree. Also, the abstract tree is infinite, but every infinite path contains a repetition node, meaning that the tree is regular and has the finite representation shown in the Figure—termination is not a problem here, because both the set of commands and the *AbsVal* set are finite.

2.1 Relating Concrete to Abstract Traces

Intuition tells us that a homomorphism should relate the concrete semantics in Figure 1 to the abstract one in Figure 2. Let $\beta : Val \to AbsVal$ map concrete data to the abstract tokens that best represent them: e.g., $\beta(2n) = e$ and $\beta(2n+1) = o$, for $n \geq 0$. Expressed in terms of the transition relation, the *homomorphism property* reads: for all program points, pp, pp', and $c, c' \in Val$,

$$\begin{array}{l}(c \vdash pp) \to (c' \vdash pp') \text{ implies there exists } a' \in AbsVal \\ \quad \text{and there exists a transition } (\beta(c) \vdash pp) \to (a' \vdash pp') \text{ such that } \beta(c') \sqsubseteq a'\end{array}$$

The inequality, $\beta(c') \sqsubseteq a'$, is a weakening of the expected $\beta(c') = a'$ because an acceptable *a.i.* can lose precision with respect to the partial ordering on *AbsVal*. For example, we might code the `div2` operation in Figure 2 so that it is deterministic: $(e \vdash \texttt{x := x div2}) \to (\top \vdash \texttt{x := succ x})$, where $\top$ represents "either even or odd." The extra element necessitates this *approximation ordering* [19] on $AbsVal = \{e, o, \top\}$: $a \sqsubseteq \top$ and $a \sqsubseteq a$, for all $a \in AbsVal$. To help prove safety, we require that the abstract transition relation is *monotonic* with respect to the ordering:

$$\begin{array}{l}(a_1 \vdash pp) \to (a_1' \vdash pp') \text{ and } a_1 \sqsubseteq a_2 \\ \quad \text{imply there exists transition } (a_2 \vdash pp) \to (a_2' \vdash pp') \text{ such that } a_1' \sqsubseteq a_2'\end{array}$$

Momentarily, we will see that existence of the homomorphism property ensures an *a.i.* is a safe simulation of its *c.i.*, but additional notations are convenient: First, define a binary relation, $safe_{Val} \subseteq Val \times AbsVal$, as

$$c \; safe_{Val} \; a \text{ iff } \beta(c) \sqsubseteq a$$

We say that c is safely approximated by a [59]. Next, define a safety relation upon the states:

$$(c \vdash pp) \; safe_{State} \; (a \vdash pp) \;\text{ iff }\; c \; safe_{Val} \; a$$

that is, a concrete state is safely approximated by an abstract state if the respective input values are related and the corresponding program points are the same pp.

For a computation tree, t, we write $root(t)$ to denote its root, and we write $t \longrightarrow t'$ to denote that there is a transition, $root(t) \rightarrow root(t')$, from t's root to a child tree, t'.

A concrete tree, t_C, is *safely simulated* (or *approximated*) by an abstract tree, t_A, iff $t_C \; safe_{Tree} \; t_A$, where

$$\begin{array}{l} t \; safe_{Tree} \; t' \text{ iff } root(t) \; safe_{State} \; root(t'), \text{ and, for every transition, } t \longrightarrow t_i, \\ \text{ there exists a transition, } t' \longrightarrow t'_j, \text{ such that } t_i \; safe_{Tree} \; t'_j \end{array}$$

Pictorially, we have

$$\begin{tikzcd}[row sep=small] root(t) \arrow[d] & safe_{State} & root(t') \arrow[d] \\ root(t_i) & & root(t'_j) \\ t_i & safe_{Tree} & t'_j \end{tikzcd}$$

The intent of $safe_{Tree}$ is that every trace path in t_C is safely simulated by one in t_A—the structure of the concrete tree is "embedded" in the abstract one. A technical issue is that the definition of $safe_{Tree}$ is recursive, and the largest such relation satisfying the recursion is desired. This motivates definition and proof by coinduction, which is discussed in the next section.

We now reach the payoff for the definitions: for every program p and input $c \in Val$, let $tree_C(p_0, c)$ be p's concrete tree, where p_0 is p's entry program point; similarly, $tree_A(p_0, a)$ is the program's abstract tree, for $a \in AbsVal$.[2] Then, $c \; safe_{Val} \; a$ implies $tree_C(p_0, c) \; safe_{Tree} \; tree_A(p_0, a)$, if the following *relational homomorphism property* holds for the concrete and abstract semantics: For program points pp, pp' and $c, c' \in Val$,

$$\begin{array}{l} c \; safe_{Val} \; a \text{ and } (c \vdash pp) \rightarrow (c' \vdash pp') \text{ imply there exists } a' \in AbsVal \\ \text{ and there exists } (a \vdash pp) \rightarrow (a' \vdash pp') \text{ such that } c' \; safe_{Val} \; a' \end{array}$$

As a picture, we have

$$\begin{tikzcd} c \vdash pp \arrow[d] & safe_{State} & a \vdash pp \arrow[d] \\ c' \vdash pp' & safe_{State} & a' \vdash pp' \end{tikzcd}$$

The relational homomophism property is easily proved equivalent to the homomorphism property given earlier.

From here on, we work with the relational representations [1, 19, 56]; alternative frameworks are discussed at length in [19]. Indeed, it is possible to begin the discussion of safety not with a β map but with a relation, $safe_{Val}$, provided that $safe_{Val}$ is *U-closed*: $c \; safe_{Val} \; a$ and $a \sqsubseteq a'$ imply $c \; safe_{Val} \; a'$. U-closure is the minimum needed to

[2] The definitions for $tree_C(p_0, c)$ and $tree_A(p_0, a)$ are in Section 3.

attempt simulation proofs. If *AbsVal* is a complete lattice and $safe_{Val}$ is *G-closed*—that is, $c\ \ safe_{Val}\ \ \sqcap\{a' \mid c\ \ safe_{Val}\ \ a'\}$—then $safe_{Val}$ defines a Galois connection between $\mathcal{P}(Val)$ and *AbsVal* [17, 41, 52][3]: $(\alpha{:}\,\mathcal{P}(Val) \rightarrow AbsVal,\ \gamma{:}\,AbsVal \rightarrow \mathcal{P}(Val))$, where $\gamma(a) = \{c \mid c\ safe_{Val}\ a\}$ and $\alpha(S) = \sqcup_{c \in S}\{\sqcap\{a \mid c\ safe_{Val}\ a\}\}$. [4] Even if *AbsVal* is not a complete lattice, G-closure is valuable, because it ensures that every $c \in Val$ has a most precise approximation in *AbsVal*.

2.2 How to Derive the Abstract Semantics from the Concrete One

A concrete semantics consists of concrete value domain(s) and rules/operators that generate transitions in concrete computation trees. Once an abstract domain, *AbsVal*, is selected, we must derive the abstract rules/operators from the concrete ones so that the relational homomorphism property holds. If one begins with a function, $\beta : Val \rightarrow AbsVal$, such that $\beta(c) \in AbsVal$ is the best approximation of $c \in Val$, (or, if one begins with a U- and G-closed relation, $safe_{Val}$, and one defines $\beta(c) = \sqcap\{a \mid c\ safe_{Val}\ a\}$), then for each concrete rule, $(c \vdash pp) \rightarrow (c' \vdash pp')$, one creates the abstract rule $(\beta(c) \vdash pp) \rightarrow (\beta(c') \vdash pp')$. In addition, the abstract rule set must be closed under monotonicity: if $(a_1 \vdash pp) \rightarrow (a_2 \vdash pp')$ is an abstract rule and $a_1 \sqsubseteq a_1'$, then there must exist a rule $(a_1' \vdash pp) \rightarrow (a_2' \vdash pp')$, such that $a_2 \sqsubseteq a_2'$. (Of course, the obvious choice is $a_2' = a_2$.)

This formulation ensures the relational homomorphism property, because every concrete rule has an abstract rule that simulates it. The even-odd analysis in Figure 2 is derived in this way. If instead, $AbsVal = \{e, o, \top\}$ is used in Figure 2, ordered as before, then transitions of the form $(\top \vdash pp) \rightarrow (a' \vdash pp')$ must be included. Augmenting Figure 2, we might add

$$(\top \vdash \texttt{x := succ x}) \rightarrow (o \vdash \texttt{even x})$$
$$(\top \vdash \texttt{x := succ x}) \rightarrow (e \vdash \texttt{even x})$$

as the transition rules from the state, $\top \vdash$ `x := succ x`. But it is also safe to replace these two transitions by just $(\top \vdash \texttt{x := succ x}) \rightarrow (\top \vdash \texttt{even x})$, and it is safe (but unenlightening) to include all three rules.

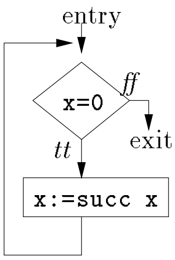
If one lacks G-closure, the abstract rules are necessarily less precise. In this case, for each concrete rule, $(c \vdash pp) \rightarrow (c' \vdash pp')$, and for every $a, a' \in AbsVal$ such that $c\ safe_{Val}\ a$ and $c'\ safe_{Val}\ a'$, one creates the abstract rule $(a \vdash pp) \rightarrow (a' \vdash pp')$. This ensures the relational homomorphism property.

2.3 Termination of the Abstract Trace

The abstract tree in Figure 2 is infinite, but its construction is finite because a state repeats in the infinite path; the tree is regular and can be represented by a finite one with backwards arc(s). Unfortunately, there is no guarantee that every abstract tree is regular: for example, a typical constant propagation analysis uses an infinite *AbsVal* set, and the *a.i.* proceeds just like its corresponding *c.i.* To terminate, constant propagation maintains a "memo table"

[3]Recall that a Galois connection is a pair of monotone functions, $(f{:}\,P \rightarrow Q, g{:}\,Q \rightarrow P)$, for complete lattices P and Q, such that $f \circ g \sqsubseteq id_Q$ and $id_P \sqsubseteq g \circ f$. The intuition is that $f(p)$ identifies p's most precise representative within Q (and similarly for $g(q)$).

[4]If one begins with $\beta : Val \rightarrow AbsVal$, then $\beta(c) = \sqcap\{a \mid c\ safe_{Val}\ a\}$, of course. Also, recall that $\sqcap S$ represents the *greatest lower bound* of S: An element, a_0, is a lower bound of S if $a_0 \sqsubseteq a$, for all $a \in S$; a_0 is the greatest lower bound if $a' \sqsubseteq a_0$, for all the lower bounds, a', of S. Dually, the *least upper bound*, $\sqcup S$, of S is that element that is the smallest of all the upper bounds of the elements of S.



Concrete Transitions:
$Val = Nat$
$(0 \vdash \mathtt{x = 0}) \rightarrow (0 \vdash \mathtt{x := succ\ x})$
$(n+1 \vdash \mathtt{x = 0}) \rightarrow (n+1 \vdash exit)$
$(n \vdash \mathtt{x := succ\ x}) \rightarrow (n+1 \vdash \mathtt{x = 0})$
Abstract Transitions:
$AbsVal = Val \cup \{\top\}$
above rules plus
$(\top \vdash \mathtt{x = 0}) \rightarrow (\top \vdash \mathtt{x := succ\ x})$
$(\top \vdash \mathtt{x = 0}) \rightarrow (\top \vdash exit)$
$(\top \vdash \mathtt{x := succ\ x}) \rightarrow (\top \vdash \mathtt{x = 0})$

Memoized tree:
$0 \vdash \mathtt{x = 0}$
$\downarrow$
$0 \vdash \mathtt{x := succ\ x}$
$\downarrow$
$1 \vdash \mathtt{x = 0}$
$0 \sqcup 1 =$
$\top \vdash \mathtt{x = 0}$
$\swarrow \qquad \searrow$
$\top \vdash exit$
$\top \vdash \mathtt{x := succ\ x}$
$0 \sqcup \top =$
$\top \vdash \mathtt{x := succ\ x}$
$\downarrow$
$\top \vdash \mathtt{x = 0}$

Figure 3: Memoized abstract interpretation

or "cache" of program points and the inputs that arrive at those points; we call this a *memoization*.

Figure 3 shows a constant propagation analysis with memoization applied to the states of the abstract tree: When the tree is computed, starting from its root node, one watches for the appearance of a new leaf, $a \vdash pp$, such that program point pp has already appeared in the path from the root to $a \vdash pp$. When this happens, the leaf's nearest ancestor of the form, $a' \vdash pp$, is located, and a' is joined to a, producing the revised leaf, $a \sqcup a' \vdash pp$. The computation proceeds. In the Figure, one sees this process employed when program point `x=0` repeats: the values 0 and 1 are joined into $\top$, giving the new state, $\top \vdash \mathtt{x = 0}$. Several transitions later this state repeats, and the infinite path need not be explored further.

Memoization ensures termination if *(i)* $AbsVal$ is partially ordered so that it is a sup-semilattice of *finite height*, that is, joins exist and there exist no infinite chains of distinct elements; and *(ii)* the state transitions are monotone on $AbsVal$. These two conditions ensure that for each program point, pp, the sequence of states, $a_i \vdash pp$, $i \geq 0$, occurring along a path forms a chain, and the finite height property ensures that the chain finishes with a repeating node. If safety has been proved for the nonmemoized *a.i.*, safety is preserved for the memoized one, since $safe_{Tree}$ is ⊔-closed.

Memoization is an instance of *widening* [17], because it generates from a starting state a regular tree that is greater than the least fixed point of the semantics rules applied to the starting state. This suggests that *narrowing* [17] might be applied to refine the regular tree, but this topic is not investigated here.

2.4 Collecting Semantics: State- and Path-Based

Once a program's tree is constructed, whether it is a concrete tree or an abstract tree, information must be extracted from it for validation or code improvement. The extracted information is called the *collecting semantics* (or "sticky semantics" [57]).

The classic collecting semantics is state based (also known as "first order"): It associates to each program point the set of input values that appeared at the program point in the tree [17, 57]: for tree t, its state-based collecting semantics, $coll_t : ProgramPoint \to \mathcal{P}(Val)$, is simply defined as

$$coll_t(pp) = \{v \mid v \vdash pp \text{ is a state in } t\}$$

In Figure 1, $coll_{t_C}(\texttt{even x}) = \{3, 4\}$, and in Figure 2, $coll_{t_A}(\texttt{even x}) = \{e, o\}$.

The term "collecting semantics" has been used traditionally for the information taken from the concrete tree, but it applies to abstract trees, also: In the data-flow-analysis literature, the collecting semantics is known as the *meet-over-all-paths* (MOP) solution [33, 47, 57].

If safety has been proved, then it follows that the collecting semantics for an abstract tree safely approximates the collecting semantics of the corresponding concrete tree, which is the "fundamental theorem" of abstract interpretation: For concrete tree, t_c, abstract tree, t_A, $t_c\ safe_{Tree}\ t_A$ implies for all $pp \in ProgramPoint$,

$$coll_{t_C}(pp) \subseteq \gamma(coll_{t_A}(pp)),$$

where $\gamma : \mathcal{P}(AbsVal) \to \mathcal{P}(Val)$ is defined $\gamma(S) = \{c \mid \text{ exists } a \in S \text{ such that } c\ safe_{Val}\ a\}$.

A proof sketch goes as follows: If $c_0 \in coll_{t_C}(pp)$, then there exists a subtree, t_0, in t_C, such that $root(t_0) = (c_0 \vdash pp)$. Since $t_C\ safe_{Tree}\ t_A$ holds, a contradiction arises unless there exists a subtree, t'_0, of t_A, such that $root(t'_0) = (a_0 \vdash pp)$ and $t_0\ safe_{Tree}\ t'_0$. Thus, $c_0\ safe_{Val}\ a_0$ holds, implying that $c_0 \in \gamma(coll_{t_A}(pp))$.

Perhaps more important is path-based ("second-order") collecting semantics, which associates to each program point the set of paths that go into or that emanate from the program point. We define the forwards and backwards path-based collecting semantics as follows:

$$\begin{aligned} fcoll_t(pp) &= \{p \mid p \text{ is a path in } t \text{ from } root(t) \text{ to some } v \vdash pp\} \\ bcoll_t(pp) &= \{p \mid p \text{ is a maximal path in } t \text{ such that } root(p) = v \vdash pp\} \end{aligned}$$

Examples of path-based collecting semantics are available-expression and live-variable data-flow analyses, which are respectively forwards and backwards, but path-based collecting semantics lie at the foundations of model-checking, as well; this application is examined later.

Finally, Cousot and Cousot [19] suggest that the collecting semantics of a tree can be any property or set of properties expressed in a logic, $\mathcal{L}$. Given tree, t, and proposition, $\phi \in \mathcal{L}$, we write $t \models \phi$ if ϕ holds true of t. For the sake of discussion, we define the collecting semantics of t to be $coll_t = \{\phi \mid t \models \phi\}$. As above, we wish to define collecting semantics of both concrete and abstract interpretations, and we assume that the same $\mathcal{L}$ can be used for both concrete and abstract traces. We must require a weak consistency property of the safety relation, $safe_{Tree}$, and $\mathcal{L}$: For all concrete trees, t_C and abstract trees, t_A:

$$t_C\ safe_{Tree}\ t_A \Rightarrow (\text{ for all } \phi \in \mathcal{L}, t_A \models \phi \Rightarrow t_C \models \phi)$$

That is, any property possessed by an abstract tree, t_A, must also hold for a corresponding concrete tree, t_C. This is the minimum needed to work confidently with $\mathcal{L}$. Next, one might desire a weakly complete relationship:

$$t_C\ safe_{Tree}\ t_A \text{ iff } (\text{ for all } \phi \in \mathcal{L}, t_A \models \phi \Rightarrow t_C \models \phi)$$

To have weak completeness, there must be a close or exact match between the primitive propositions of $\mathcal{L}$ and *AbsVal* [19].

The above two notions are titled "weak" because decidability is lacking: t_C $safe_{Tree}$ t_A and $t_A \not\models \phi$ do *not* imply $t_C \not\models \phi$. If one replaces the rightmost $\Rightarrow$ in the definitions above by *iff*, one obtains strong consistency and strong completeness, respectively. The strong versions of the definitions give decidability and can be put to good use on specific case studies, but they appear difficult to realize in general.

These notions of soundness and completeness are developed by Dams in his thesis [22]. Finally, Bruns [6] and Levi [50] suggest that one might work profitably with different logics $\mathcal{L}_C$ and $\mathcal{L}_A$ for the concrete and abstract interpretations, respectively: for $\phi \in \mathcal{L}_C$, $\phi' \in \mathcal{L}_A$, t_C $safe_{Tree}$ t_A and ϕ $safe_{Prop}$ ϕ' imply ($t_C \models \phi$ iff $t_A \models \phi'$).

2.5 Representations of the Collecting Semantics

A typical implementation of an *a.i.* sidesteps the generation of a program's abstract computation tree and calculates directly the tree's collecting semantics. This is done by computing upon a set of equations or constraints that defines the collecting semantics, one equation/constraint per program point; solution of the equations/constraints yields the collecting semantics. Examples of such representations of the collecting semantics are the table generated from solving a set of data flow equations (see the next section); the cache generated from solving a set of denotational semantics equations [36, 14]; and the solution of a constraints set generated for type inference [4, 5, 82] or control-flow analysis [34, 64].[5]

Because of the emphasis placed upon collecting semantics, it is easy to confuse an abstract computation tree with the collecting semantics extracted from it. As a result, precision can be lost inadvertantly when one formulates an algorithm for calculating directly the collecting semantics rather than the abstract tree from which the collecting semantics is extracted. Also, safety proofs become complicated when they are worked on the collecting semantics algorithm rather than upon an algorithm that generates the abstract tree.

Our recommendation is that an algorithm for calculating the collecting semantics should be defined and proved safe with respect to the *a.i.* upon which it is based.

2.6 Application: Data-Flow Analysis

As an example of direct computation of collecting semantics, consider a standard iterative data-flow analysis, which encodes a program and its data flow as a set of simultaneous equations, one equation per program point. The equations are solved with a fixed-point iteration [3], using an *AbsVal* set that is a finite height, pointed, sup-semilattice. The resulting solution, called the MFP ("maximal fixed point") solution [33, 47, 57], can be less precise than the MOP solution but equals it when the analysis is distributive[6] [47].

One extracts a flow-analysis equation set from the set of abstract transition rules as follows: Each rule, $(a \vdash pp) \rightarrow (a' \vdash pp')$, defines a monotone "transfer function":

$$f_{a,pp,a',pp'}(a'') = \text{ if } a \sqsubseteq a'' \text{ then } a' \text{ else } \perp$$

[5] Contrast this with the classic formulation of strictness analysis [8], which is a true *a.i.* and *not* a calculation of a collecting semantics.

[6] The flow analysis is distributive when all "transfer functions," f, used in the flow equations are distributive: $f(x \sqcup y) = f(x) \sqcup f(y)$, for all $x, y \in AbsVal$.

$$
\begin{aligned}
in_{\texttt{even x}} &= f_{e,\texttt{x:=succ x},o,\texttt{even;x}}(in_{\texttt{x:=succ x}}) \sqcup f_{o,\texttt{x:=succ x},e,\texttt{even;x}}(in_{\texttt{x:=succ x}}) \\
&\quad \sqcup f_{\top,\texttt{x:=succ x},o,\texttt{even;x}}(in_{\texttt{x:=succ x}}) \sqcup f_{\top,\texttt{x:=succ x},e,\texttt{even;x}}(in_{\texttt{x:=succ x}}) \sqcup e \\
in_{\texttt{x:=x div2}} &= f_{e,\texttt{even x},e,\texttt{even;x}}(in_{\texttt{even x}}) \sqcup f_{\top,\texttt{even x},e,\texttt{even;x}}(in_{\texttt{even x}}) \\
in_{\texttt{x:=succ x}} &= f_{e,\texttt{x:=x div2},e,\texttt{x:=succ;x}}(in_{\texttt{x:=x div2}}) \sqcup f_{e,\texttt{x:=x div2},o,\texttt{x:=succ;x}}(in_{\texttt{x:=x div2}}) \\
&\quad \sqcup f_{o,\texttt{x:=x div2},e,\texttt{x:=succ;x}}(in_{\texttt{x:=x div2}}) \sqcup f_{o,\texttt{x:=x div2},o,\texttt{x:=succ;x}}(in_{\texttt{x:=x div2}}) \\
&\quad \sqcup f_{\top,\texttt{x:=x div2},e,\texttt{x:=succ;x}}(in_{\texttt{x:=x div2}}) \sqcup f_{\top,\texttt{x:=x div2},o,\texttt{x:=succ;x}}(in_{\texttt{x:=x div2}}) \\
in_{exit} &= f_{o,\texttt{even x},o,exit}(in_{\texttt{even x}}) \sqcup f_{\top,\texttt{even x},o,exit}(in_{\texttt{even x}})
\end{aligned}
$$

iteration	0	1	2	3	4	5	6
$in_{\texttt{even x}}$	$\bot$	e	e	e	$\top$	$\top$	$\top$
$in_{\texttt{x:=x div2}}$	$\bot$	$\bot$	e	e	e	e	e
$in_{\texttt{x:=succ x}}$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$
in_{exit}	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	o	o

Figure 4: Iterative data-flow analysis

Therefore, if $f_{a,pp,a',pp'}(a_0) \neq \bot$, then there exists a transition from $a_0 \vdash pp$ to pp'.

Next, for each program point, pp', we define its flow equation, $in_{pp'} = \cdots$, in terms of the join of the transfer functions that produce input values for pp':

$$in_{pp'} = \bigsqcup\{f_{a,pp,a',pp'}(in_{pp}) \mid pp \in ProgramPoint, a, a' \in AbsVal\}$$

Finally, to model the start state, $a_0 \vdash pp_0$, one joins the initial input value, a_0, to the flow equation for pp_0: $in_{pp_0} = \cdots \sqcup a_0$. The resulting equation set is solved with the usual fixed-point iteration.

As an example, we convert the *a.i.* in Figure 2 into the flow analysis in Figure 4—note that *AbsVal* must be extended with $\top$ and $\bot$ values so that it is a pointed sup-semilattice; the partial ordering on *AbsVal* is the expected one. Next, the abstract rules in Figure 2 are extended monotonically for the new values, $\top$ and $\bot$: $(\top \vdash pp) \rightarrow (a' \vdash pp')$ is added for each rule $(a \vdash pp) \rightarrow (a' \vdash pp')$ in Figure 2. (No rules are needed for states of form $pp \vdash \bot$.) The resulting flow equations in Figure 4 compute the iterates of the fixed-point solution, displayed in the table, for the program with the starting state $e \vdash$ `even x`.

For a given program, its start state, and its abstract semantics, let t_A be the program's (potentially nonregular) abstract tree built without memoization, let tm_A be its (regular) tree built with memoization, and let $\{in_{pp} \mid pp \in ProgramPoint\}$ be the least-fixed point solution of the flow equations induced from the program. With standard techniques [47], one can prove the standard results: for all $pp \in ProgramPoint$, *(i)* $coll_{t_A}(pp) \subseteq coll_{tm_A}(pp) \subseteq in_{pp}$; *(ii)* when the flow analysis is distributive, $coll_{t_A}(pp) = coll_{tm_A}(pp) = in_{pp}$. That is, the MOP solutions on the unmemoized and memoized trees are at least as precise as the MFP one and are equal for distributive analyses. Examples exist where the subset inclusions in *(i)* are proper.

Many flow analyses—available expressions and live variables, for example—are path based, because the analyses must calculate execution paths containing histories of expression evaluation and futures of variable use. The implementations of the analyses compute representations of the paths, namely, sets of available expressions and sets of live variables. Path-based data-flow analyses are intimately related to model checking, which we now examine.

$$\phi \in \mathit{Proposition} \qquad p \in \mathit{PrimitiveProp} \qquad Z \in \mathit{Iden}$$

$$\phi \ ::= \ p \ | \ \neg p \ | \ \phi_1 \wedge \phi_2 \ | \ \phi_1 \vee \phi_2 \ | \ \Box\phi \ | \ \Diamond\phi \ | \ \mu Z.\phi \ | \ \nu Z.\phi \ | \ Z$$

Let State be the states of a trace, and let $\rho \in \mathit{PEnv} = \mathit{Iden} \to \mathcal{P}(\mathit{State})$. Define $[\![\cdot]\!] \in \mathit{Proposition} \to \mathit{PEnv} \to \mathcal{P}(\mathit{State})$ as

$$
\begin{array}{l}
[\![p]\!]\rho = \{s \mid s \models p\} \quad \text{Note: this set must be given at the outset.} \\
[\![\neg p]\!]\rho = \{s \mid s \not\models p\} \\
[\![\phi_1 \wedge \phi_2]\!]\rho = [\![\phi_1]\!]\rho \cap [\![\phi_2]\!] \\
[\![\phi_1 \vee \phi_2]\!]\rho = [\![\phi_1]\!]\rho \cup [\![\phi_2]\!] \\
[\![\Box\phi]\!]\rho = \{s \mid \text{ for all } s' \text{ such that } s \to s', s' \in [\![\phi]\!]\rho\} \\
[\![\Diamond\phi]\!]\rho = \{s \mid \text{ there exists } s' \text{ such that } s \to s' \text{ and } s' \in [\![\phi]\!]\rho\} \\
[\![\mu Z.\phi]\!]\rho = \bigcup_{i \geq 0} S_i, \text{ where } \left\{ \begin{array}{l} S_0 = \emptyset \\ S_{i+1} = [\![\phi]\!]([Z \mapsto S_i]\rho) \end{array} \right. \\
[\![\nu Z.\phi]\!]\rho = \bigcap_{i \geq 0} S_i, \text{ where } \left\{ \begin{array}{l} S_0 = \mathit{State} \\ S_{i+1} = [\![\phi]\!]([Z \mapsto S_i]\rho) \end{array} \right. \\
[\![Z]\!]\rho = \rho(Z)
\end{array}
$$

For a closed proposition, ϕ, define $s \models \phi$ iff $s \in [\![\phi]\!]\{\}$.

Figure 5: Modal mu-calculus syntax and semantics

2.7 Application: Model Checking

Model checking is a technique for validating properties of paths in a program's computation tree [9, 22, 51]. The technique is used primarily to validate safety and liveness properties of circuits and protocols, but it is applicable to validating properties of concrete and abstract computation trees [10, 72, 76, 77].

Properties are stated in a logic, $\mathcal{L}$, of which CTL* [9] and modal mu-calculus [79] are commonly used; we employ the latter. Figure 5 defines the syntax and semantics of the modal mu-calculus. The two modal operators are central: $\Box\phi$ holds true at a state, s, in a trace, written $s \models \Box\phi$, if all one-step transitions from s go to states, s' such that $s' \models \phi$. Similarly, $s \models \Diamond\phi$ if there exists a transition from s to a successor state, s', such that $s' \models \phi$. Properties that span paths longer than one transition are conveniently coded by the recursion operators, μ and ν;[7] for example, to state that ϕ holds true for every state in every path (including the infinite ones) from the current state, one writes $\nu Z.\phi \wedge \Box Z$, and to assert that ϕ must hold true at a state located some finite distance from the current one, one writes $\mu Z.\phi \vee \Diamond Z$.

The tree in Figure 2 can be model checked for simple path properties—for example, one can verify that all paths from the root must include the command `x:=succ x` by checking the proposition $\mu Z.(pp = (\mathtt{x} := \mathtt{succ\ x})) \vee \Box Z$, where pp denotes the value of the program point at a state in the trace. One can check if a state may lead to termination

[7] The semantics of μ and ν are stated as countably infinite union and intersection respectively, because the traces that we use are forwards and backwards *image finite*: the sets of successor and predecessor states for every state in the trace are finite.

via $\mu Z.(pp = \texttt{exit}) \vee \Diamond Z$, and this proposition appears to be true for the root, but this is *unsound*: because an *a.i.* adds extra execution paths, it might add one that leads to an exit, where no such path exists in the corresponding *c.i.* (Consider the *c.i.* for the program in Figure 2 with input 2.)

Model checking upon a safe *a.i.* is weakly complete when the $\Diamond$ operator is removed from the calculus; call the result the *box-mu-calculus*.[8]

Since it is a language of path properties, the box mu-calculus is ideal for expressing second-order (path-based) collecting semantics. For tree, t, one might define $coll_t = \{\phi \mid t \models \phi\}$. But t contains states of the form $a \vdash pp$, and it is traditional to define collecting semantics in terms of the properties that hold at the tree's program points: $coll_t(pp) = \{\phi \mid \text{for every subtree}, t' \text{ in } t, \text{ such that } root(t') = a \vdash pp \text{ and } t' \models \phi\}$.

With this collecting semantics definition, one can encode a path-based data-flow analysis as a proposition in the modal mu-calculus [24, 72, 76, 77]; the propositions are model checked on an *a.i.* where $AbsVal = \{\bullet\}$ and $c\ safe_{Val}\ \bullet$ holds for all $c \in Val$—of course, this is the program's control flow graph. When the nodes of the control flow graph are annotated with local information (*gen*-, and *kill*-sets), the model check effectively propagates the local information through the nodes of the graph, like a data-flow analysis does.

For example, the flow equations for very busy expressions analysis [48] have format

$$VBE_{pp} = UsedIn_{pp} \cup (NotMod_{pp} \cap (\bigcap_{q \in succ\ pp} VBE_q))$$

which calculates the set of expressions that must be used at some point in the future from program point, *pp*. The flow equations are solved with a greatest fixed point calculation: The initial approximation to the solution consists of sets of all the expressions in the program, and iteration of the equations on the initial approximation trims the sets down to size.

The above flow equation format translates to a mu-calculus proposition that asks whether a specific expression, e, is very busy at a program point:

$$isVBE_e = \nu Z.IsUsed_e \vee (\neg IsModified_e \wedge \Box Z)$$

Based on the local information, $IsUsed_e$ and $IsModified_e$ for each flowchart box, the model checker attempts to validate the proposition for the nodes of the control flow graph—the model checker is the "engine" for calculating data flow.[9]

Rather than working with the control flow graph, one can obtain a higher precision model check by working with a less trivial *a.i.* of a program. For example, Clarke, Grumberg, and Long use this technique for circuit and protocol validation [10] by model checking abstract interpretations based on *modulo-n* arithmetic.

There is a correspondence in the other direction as well: The standard algorithm for checking CTL (or modal mu-calculus without alternating fixed-point quantifiers) translates a CTL proposition into a first-order flow equation set and solves iteratively [27].

[8]The proof of weak completeness for the box-mu-calculus requires that weak completeness holds for *PrimitiveProp*.

[9]Note that the mu-calculus formula for computing live variables is coded $islive_x = \mu Z.IsUsed_x \vee (\neg IsKilled_x \wedge \Diamond Z)$, which is an *unsound* proposition to check with a safe *a.i.* In practice, the information gleaned from a live variable analysis is in fact used to detect dead variables, where $isdead_x = \neg islive_x$, which *is* a sound proposition to model check.

3 Inductively and Coinductively Defined Sets

The computation trees in the previous section can be infinite, and proofs on infinite trees are best worked with coinductive techniques, which we must review. The following presentation draws from [20].

We begin with the classical definition of induction. Let $\mathcal{U}$ be a universe of terms, and let $F\colon \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ be continuous[10] with respect to the complete lattice $\langle \mathcal{P}(\mathcal{U}), \subseteq \rangle$. 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, 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 $Q \subseteq \mathit{gfp}F$, it suffices to find a set, Q', such that $Q \subseteq Q'$ and $\mathit{dense}_F Q'$. When a property, P, is defined coinductively as $P = \mathit{gfp}F$, then proving $\mathit{dense}_F(\mathit{gfp}G)$ is a standard way of proving that coinductively defined set $\mathit{gfp}G$ has P.

Here are brief examples. Let $\mathcal{U}$ be a universe of strings of at most countably infinite (ω-) length; the BNF rule, $V ::= 0 \mid 1V$ generates the continuous functional $\bar{V} : \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$; $\bar{V}(S) = \{0\} \cup \{1s \mid s \in S\}$; we obtain $\mathit{lfp}\bar{V} = \{1^n 0 \mid n \geq 0\}$, whereas $\mathit{gfp}\bar{V} = \mathit{lfp}\bar{V} \cup \{1^\omega\}$.

It is helpful to think of strings as trees with a single path; when calculating $\mathit{lfp}\bar{V}$, S_i contains trees of height i or less that are certified members of $\mathit{lfp}\bar{V}$; in contrast, T_i contains trees that are certified to depth i and 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. In contrast, a fixed-point coinduction typically involves recursively defined predicates: say that we wish to show, for all strings (trees) in $\mathit{gfp}\bar{V}$, that no 1 follows a 0. Define these predicates:

$$\begin{array}{l} \mathit{ok}(s) \text{ iff } \mathit{zeroes}(s) \text{ or } s = 1 \text{ or } (s = 1t \text{ and } \mathit{ok}(t)) \\ \quad \text{where } \mathit{zeroes}(s) \text{ iff } s = 0 \text{ or } (s = 0t \text{ and } \mathit{zeroes}(t)) \end{array}$$

These predicates are circular, so consider the corresponding functionals: $\mathit{ok}'(P) = \{s \mid (\mathit{gfp}\ \mathit{zeroes}'(s)) \text{ or } s = 1 \text{ or } (s = 1t \text{ and } t \in P)\}$, $\mathit{zeroes}'(Q) = \{s \mid s = 0 \text{ or } (s = 0t \text{ and } t \in Q)\}$, and define $Ok = \mathit{gfp}\ \mathit{ok}'$. (This ensures that $1^\omega \in Ok$, for example.) To prove $\mathit{gfp}\bar{V} \subseteq Ok$, it suffices to prove $\mathit{dense}_{ok'}\ \mathit{gfp}\bar{V}$, which requires the trivial lemma that $\mathit{dense}_{zeroes'}\{0\}$.

For the remainder of this paper, we use a universe, $\mathcal{U}$, of finitely branching trees of at most countably infinite (ω-) depth[29, 31].

[10] A monotone function would suffice, but continuity ensures fixed point convergence by the first limit ordinal. Recall that function F is continuous if $F(\sqcup S) = \sqcup\{F(s) \mid s \in S\}$, for all sets (*chains*) $S = \{s_0, s_1, \ldots, s_i, \ldots\}$, such that $s_0 \sqsubseteq s_1 \sqsubseteq \cdots \sqsubseteq s_i \sqsubseteq \cdots$.

3.1 Coinduction Applied to Concrete and Abstract Interpretations

A computation tree is an element of a (co)inductively defined set, which we now define. Here is the specification of a well formed tree (*wft*):

1. $v \vdash pp$ is a *wft*;

2. If $\{(v \vdash pp) \rightarrow (v_i \vdash pp_i)\}_{i \in I}$ is the set of *all* possible transitions from state $v \vdash pp$, and for each i, t_i is a *wft* such that $root(t_i) = (v_i \vdash pp_i)$, then $(v \vdash pp) \longrightarrow \{t_i\}_{i \in I}$ is a *wft*.

When the above definition is interpreted inductively, the well-formed trees are the finite ones; a coinductive interpretation includes the countably infinite-depth trees. We use the coinductive interpretation.

If the relational homomorphism property holds, we can prove safe simulation: Let wft_C and wft_A denote the sets of well-formed concrete and abstract trees, respectively. We have this result, for all $t_0 \in wft_C$ and $t'_0 \in wft_A$:

$$\text{if } root(t_0)\ safe_{State}\ root(t'_0), \text{then } t_0\ safe_{Tree}\ t'_0$$

The proof proceeds as follows: First, note that $safe_{Tree} = gfpF(S) = \{(t,t') \mid t \in wft_C, t' \in wft_A, root(t)\ safe_{State}\ root(t')$ and for all $t \longrightarrow t_i$, there exists $t' \longrightarrow t'_j$ such that $S(t_i, t'_j)\}$. Now, consider the set $S_0 = \{(t,t') \mid t \in wft_C, t' \in wft_A$, and $root(t)\ safe_{State}\ root(t')\}$. We know that $(t_0, t'_0) \in S_0$, so the result we desire will follow from the proof that $S_0 \subseteq F(S_0)$. This goes as follows: For $(t,t') \in S_0$, when $t \longrightarrow t_i$, where $t_i \in wft_C$ and $root(t_i) = (c_i \vdash p_i)$, there must exist a transition $root(t') \rightarrow a_j \vdash p_i$ such that $c_i\ safe_{Val}\ a_j$ by the relational homomorphism property. Since $t' \in wft_A$, $a_j \vdash p_i$ must be the root of some tree $t'_j \in wft_A$, implying that $t' \longrightarrow t'_j$. Finally, it is immediate that $(t_i, t'_j) \in S_0$. (Indeed, one can prove the stronger result that $S_0 = F(S_0)$.)

3.2 A Comparison with Mathematical Induction

It is useful to compare the above proof to one done by induction on the length of an execution trace—consider deterministic traces (sequences) only and an arbitrary safety relation, $\mathcal{R}$. The claim that concrete trace $t_C = C_0 \rightarrow C_1 \rightarrow \cdots \rightarrow C_i \rightarrow \cdots$ is simulated by abstract trace $t_A = A_0 \rightarrow A_1 \rightarrow \cdots \rightarrow A_i \rightarrow \cdots$ can be defined as $\forall i \geq 0, C_i\ \mathcal{R}\ A_i$. Then, a proof by induction goes in two steps:

- $C_0\ \mathcal{R}\ A_0$
- $C_i\ \mathcal{R}\ A_i$ implies $C_{i+1}\ \mathcal{R}\ A_{i+1}$

When the result is proved with coinduction techniques, the above two steps reappear, but some startup machinery is required: The universally quantified safety property is recoded recursively as $safe = gfpF$, where $F(S) = \{(t,t') \mid head(t)\ \mathcal{R}\ head(t')$ and $(tail(t), tail(t')) \in S\}$. The usual difficulty in the coinductive proof is selecting the set to be proved dense for F, but a standard choice focusses upon the heads of the traces: $S_0 = \{(t,t') \mid head(t)\ \mathcal{R}\ head(t')\}$. First, we must show that $(t_C, t_A) \in S_0$; this is the "basis step." Next, we must show that $S_0 \subseteq F(S_0)$; this is the "induction step," because it quickly decomposes to using $head(t)\ \mathcal{R}\ head(t')$ to prove $head(tail(t))\ \mathcal{R}\ head(tail(t'))$.

Although the above explanation was meant to emphasize the similarities between mathematical induction and coinduction proof techniques, we must note that the primary distinction between the two techniques is that the former decomposes traces into their component states whereas the latter handles the traces as whole entities. As tree structures and their properties grow in complexity, it becomes more convenient to work with coinduction—safety properties stay simple and proofs stay short.

3.3 Computing the Computation Tree

Although the coinductive definitions *define* the sets of well-formed computation trees, it is traditional to *compute* the trees by beginning with a start state, $v_0 \vdash p_0$, and generating incrementally the transitions that proceed from the start state. For example, we might compute this sequence of partial trees, t_i, starting from $t_0 = e \vdash \texttt{even x}$ as follows:

$$\begin{tikzcd}[column sep=small]
e \vdash \texttt{even x} & \Longrightarrow & e \vdash \texttt{even x} \arrow[d] & \Longrightarrow & & e \vdash \texttt{even x} \arrow[d] & & \Longrightarrow & \cdots \\
& & e \vdash \texttt{x := x div2} & & & e \vdash \texttt{x := x div2} \arrow[dl] \arrow[dr] & & & \\
& & & & o \vdash \texttt{x := succ x} & & e \vdash \texttt{x := succ x} & &
\end{tikzcd}$$

Which tree does this sequence denote? Recall that $wft_A = gfpF_A = \bigcap_{i \geq 0} T_i$, where $T_0 = \mathcal{U}$, $T_{i+1} = F_A(T_i)$, and F_A is the cocontinuous functional derived from the recursive definition of *wft*. Say that each partial tree, t_i, in the above sequence defines the set of trees that share the same prefix as t_i: $S_i = \{t \in T_i \mid t_i \text{ is a prefix of } t\}$. Of course, $S_\infty = \bigcap_{i \geq 0} S_i \subseteq wft_A$, and more importantly, S_∞ is a singleton set, because the definition of F_A demands *all* possible transitions from a state to its successor states be included for each node in the tree. In this sense, the sequence of partial trees, t_i, computed from a start state, $v_0 \vdash p_0$, defines a unique tree in wft_A, and we name this tree $tree(p_0, v_0)$.

Both the concrete and abstract trees of a program can be computed this way, and we have this corollary: for $c \in Val$, $a \in AbsVal$, $c\ safe_{Val}\ a$ implies $tree(p_0, c)\ safe_{Tree}\ tree(p_0, a)$.

4 Liveness Abstract Interpretations

The examples thus far are oriented towards safety analyses, where an *a.i.* generates more paths in its abstract trees than does the corresponding *c.i.* A *liveness analysis* is the dual: An abstract tree contains a transition only if all corresponding concrete trees possess a corresponding transition (cf. *conservative transitions*, e.g., [11]). Liveness analyses are of primary interest when one wishes to validate properties such as starvation freedom.

As before, one defines an abstract value set, *AbsVal*, and a binary relation, $live_{Val} \subseteq Val \times AbsVal$; a relation, $live_{State}$, must be defined so that the liveness relation on trees is expressed as follows:

$$\begin{array}{l}
t\ live_{Tree}\ t' \text{ iff } root(t)\ live_{State}\ root(t'), \text{ and} \\
\quad \text{for every transition, } t' \longrightarrow t'_j, \\
\quad \text{there exists a transition, } t \longrightarrow t_i, \text{ such that } t_i\ live_{Tree}\ t'_j
\end{array}$$

That is, the *c.i.* is a simulation of the *a.i.* To prove the liveness relation, the (dual of the) relational homomorphism property is required.

$$op \in PrimitiveOperation$$
$$e \in Expression$$
$$f, x \in Id$$

$$e \ ::= \ op(e) \ \mid \ \texttt{if}\ e_1\ e_2\ e_3 \ \mid \ \texttt{rec}\ f\ x.\ e \ \mid \ e_1\ e_2 \ \mid \ x$$

$$v \in Val = Nat \cup Bool \cup Clos$$
$$\langle \rho, x, f, e \rangle \in Clos = Env \times Id \times Id \times Expression$$
$$\rho \in Environment = Id \overset{fin}{\to} Val$$

$$\frac{\rho \vdash e \Downarrow v}{\rho \vdash op(e) \Downarrow f_{opC}(v)} \qquad \frac{\rho \vdash e_1 \Downarrow tt \quad \rho \vdash e_2 \Downarrow v}{\rho \vdash \texttt{if}\ e_1\ e_2\ e_3 \Downarrow v} \qquad \frac{\rho \vdash e_1 \Downarrow f\!f \quad \rho \vdash e_3 \Downarrow v}{\rho \vdash \texttt{if}\ e_1\ e_2\ e_3 \Downarrow v}$$

$$\rho \vdash \texttt{rec}\ f\ x.\ e \Downarrow \langle \rho, f, x, e \rangle \qquad \rho \vdash x \Downarrow \rho(x)$$

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

Figure 6: Concrete big-step semantics

5 Analysis of Big-Step Semantics

Flowchart models break down when higher-order procedural languages and other language paradigms arise, and we must rely upon more modern forms of operational semantics. We begin with big-step (*natural*) semantics [46, 63], where a language's semantics is the set of derivations generated inductively from a set of inference rule schemes.

Big-step semantics is a second example of a computation-tree-based semantics, where a path in a big-step tree presents the computation steps that isolate and evaluate a subphrase in a source program—branching represents not necessarily nondeterminism but the distinct computations of distinct subphrases. Nonetheless, we will see that trace-based *a.i.* techniques let one develop abstract semantics, abstract trees, safe simulations, collecting semantics, and termination strategies. In addition, a coinductive interpretation of a big-step semantics lets one use infinite computation trees to model divergent programs.

Figure 6 gives the concrete semantics of an untyped, higher-order functional language where user-defined abstractions are recursive.[11] With big-step semantics, one defines the language's operations by writing one or more inference rules of the form, $\frac{s_1 \ s_2 \ \cdots \ s_n}{\rho \vdash e \Downarrow v}$, $n \geq 0$, for each source language construct, e. Of course, one reads the sequent, $\rho \vdash e \Downarrow v$, as stating, "with environment, ρ, expression e converges to v." The Figure shows that primitive operators, op, are associated with functions, f_{opC}, on nonclosure elements of Val. User-defined abstractions are packaged into closures, which are interpreted upon invocation.

A natural semantics is attribute grammar-like, because its inherited attributes sit to the left of the turnstile in a sequent, and its synthesized attributes sit to the right of the down-

[11]The problems addressed in this section are not unique to functional languages; a while-loop language with procedures behaves similarly [63].

let $p =$ `(rec f x.if even x 1 f(x div2))`

let $cl = \langle \rho, \mathtt{f}, \mathtt{x}, \mathtt{if\ even\ x} \ldots \rangle$

let $\rho_i = [\mathtt{f} \mapsto cl, \mathtt{x} \mapsto i]\rho$

- $\rho \vdash p\ \mathtt{5} \Downarrow 1$
  - $\rho \vdash p \Downarrow cl$
  - $\rho \vdash \mathtt{5} \Downarrow 5$
  - $\rho_5 \vdash \mathtt{if} \ldots \Downarrow 1$
    - $\rho_5 \vdash \mathtt{even\ x} \Downarrow \mathit{ff}$
      - $\rho_5 \vdash \mathtt{x} \Downarrow 5$
    - $\rho_5 \vdash \mathtt{f(x\ div2)} \Downarrow 2$
      - $\rho_5 \vdash \mathtt{f} \Downarrow cl$
      - $\rho_5 \vdash \mathtt{x\ div2} \Downarrow 2$
        - $\rho_5 \vdash \mathtt{x} \Downarrow 5$
      - $\rho_2 \vdash \mathtt{if} \ldots \Downarrow 1$
        - $\rho_2 \vdash \mathtt{even\ x} \Downarrow tt$
          - $\rho_2 \vdash \mathtt{x} \Downarrow 2$
        - $\rho_2 \vdash \mathtt{1} \Downarrow 1$

Note:

$f_{\mathtt{even}}(2n) = tt$
$f_{\mathtt{even}}(2n+1) = \mathit{ff}$
$f_{\mathtt{div2}}(2n) = n$
$f_{\mathtt{div2}}(2n+1) = n$

Figure 7: Concrete interpretation of derivation

pointing arrow. Figure 7 shows the concrete computation tree of a convergent program that uses two primitive operators, `even` and `div2`, whose interpretations are given in the Figure.

Figure 8 gives an abstract semantics for even-odd analysis for the language in Figure 6. We define an abstract version of *Val*, called *AbsVal*, and we revise the operations (inference rules) to compute on $\mathit{AbsVal} = \{\mathbf{e}, \mathbf{o}\}$.

Ideally, the modifications upon the inference rules are minor—one reinterprets each f_{opC} into an f_{opA} and the remaining rules stay the same. Alas, problems arise with nondeterminism: For example, if the language possessed the rules $\dfrac{\rho \vdash e_1 \Downarrow v_1}{\rho \vdash e_1\ \mathtt{or}\ e_2 \Downarrow v_1}$ and $\dfrac{\rho \vdash e_2 \Downarrow v_2}{\rho \vdash e_1\ \mathtt{or}\ e_2 \Downarrow v_2}$, then a concrete tree for e_1 `or` e_2 would use just one of the rules, but a safe *a.i.* must employ *both*. This suggests that a program's *a.i.* might be a set of derivations, but it is traditional to code a single, nondeterministic, abstract computation tree. This forces us to join the synthesized attributes, v_1 and v_2, in effect generating a new rule scheme for the *a.i.* : $\dfrac{\rho \vdash e_1 \Downarrow v_1 \quad \rho \vdash e_1 \Downarrow v_1}{\rho \vdash e_1\ \mathtt{or}\ e_2 \Downarrow v_1 \sqcup v_2}$. This issue arises again with `if`: when its test, e_1, cannot be resolved to tt or ff, then both e_2 and e_3 must be interpreted and their values joined.[12]

Figure 9 displays the abstract tree of the example program. It is an infinite (but regular) derivation tree, which is problematic, because the standard, inductive interpretation of natural semantics prohibits infinite derivations—therefore we must interpret the abstract semantics coinductively. Also, the synthesized attribute, a, for the repeated state, $\rho_\top \vdash$ `if`... $\Downarrow a$, is unresolved. The equality $a = \mathbf{o} \sqcup a$ must be satisfied, which suggests that

[12]This raises the issue of the approximation ordering on *AbsVal*. In the case of Figure 8, the definitions of the four sets are well founded, so the sets can be defined as the smallest ones that satisfy the equations. The approximation ordering is defined in the obvious way: *AbsNat* is defined discretely; *AbsVal* is the (disjoint) union of its three components, where the orderings of the components are preserved, plus the extra element, $\top$, such that $a \sqsubseteq \top$, for all $a \in \mathit{AbsVal}$; the ordering on *AbsEnv* is pointwise; and *AbsClos*'s ordering is defined componentwise (*Id* and *Expr* are ordered discretely).

$$v \in AbsVal = AbsNat \cup Bool \cup AbsClos \cup \{\top\}$$
$$\quad \text{such that } v \sqsubseteq \top, \text{ for all } v \in AbsVal$$
$$n \in AbsNat = \{\mathbf{e}, \mathbf{o}\}$$
$$\langle \rho, x, f, e\rangle \in AbsClos = AbsEnv \times Id \times Id \times Expression$$
$$\rho \in AbsEnv = Id \overset{fin}{\to} AbsVal$$

Semantics rules for `if`, `rec`, $(e_1\, e_2)$, and x carry over from Figure 6. Alter the rule for *op* and add one rule for `if` as follows:

$$\frac{\rho \vdash e \Downarrow v}{\rho \vdash op(e) \Downarrow f_{opA}(v)} \qquad \frac{\rho \vdash e_1 \Downarrow \top \quad \rho \vdash e_2 \Downarrow v_2 \quad \rho \vdash e_3 \Downarrow v_3}{\rho \vdash \texttt{if}\ e_1\ e_2\ e_3 \Downarrow v_2 \sqcup v_3}$$

Figure 8: Abstract big-step semantics

the approximation ordering on *AbsVal* be used to calculate the least such a that satisfies the equation. More precisely, we desire the least derivation tree that satisfies the regular tree schema in the figure. We tackle these issues in turn, after first reviewing the notion of safety.

5.1 Safety Properties of Finite Derivations

For the moment, we assume that both concrete and abstract semantics are defined inductively; we handle coinductive derivations in the next section. For a universe, $\mathcal{U}$, of finitely-branching, countably-infinite-depth trees, the set of well-formed derivation trees derived from a set of inference rule instances, $\mathcal{R}$[13], is the least set satisfying the predicate $wftree_{\mathcal{R}} \subseteq \mathcal{U}$:

$$wftree_{\mathcal{R}}(t) \text{ iff there exists } \frac{s_1, \cdots, s_n}{root(t)} \in \mathcal{R}, n \geq 0,$$
$$\text{and for all child subtrees, } t_i, i \in 1 \ldots n, \text{ of } t, root(t_i) = s_i \text{ and } wftree_{\mathcal{R}}(t_i)$$

A safety relation is defined to relate the concrete and abstract intepretations, and we begin with the safety relation for the value sets, which is defined for the example as

- $v\ safe_{Val}\ \top$, for all $v \in Val$;
- $2n\ safe_{Val}\ \mathbf{e}$ and $2n+1\ safe_{Val}\ \mathbf{o}$, for $n \geq 0$;
- $tt\ safe_{Val}\ tt$ and $ff\ safe_{Val}\ ff$;
- $\langle \rho_C, f, x, e\rangle\ safe_{Val}\ \langle \rho_A, f, x, e\rangle$ iff $\rho_C\ safe_{Env}\ \rho_A$;
- $\rho_C\ safe_{Env}\ \rho_A$ iff $domain(\rho_C) = domain(\rho_A)$ and for all $i \in domain(\rho_C), \rho_C(i)\ safe_{Val}\ \rho_A(i)$

Note that $safe_{Val}$ is U-closed, which is required. Of course, the relational homomorphism property must hold for corresponding primitive operations f_{opC} and f_{opA}: if $c\ safe_{Val}\ a$, then $f_{opC}(c)\ safe_{Val}\ f_{opA}(a)$.

[13] As is the custom, $\mathcal{R}$ is a set of rule instances, rather than rule schemes.

let $p =$ `(rec f x.if even x 1 f(x div2))`
let $cl = \langle \rho, \mathtt{f}, \mathtt{x}, \mathtt{if\ even\ x} \ldots \rangle$
let $\rho_i = [\mathtt{f} \mapsto cl, \mathtt{x} \mapsto i]\rho$ in

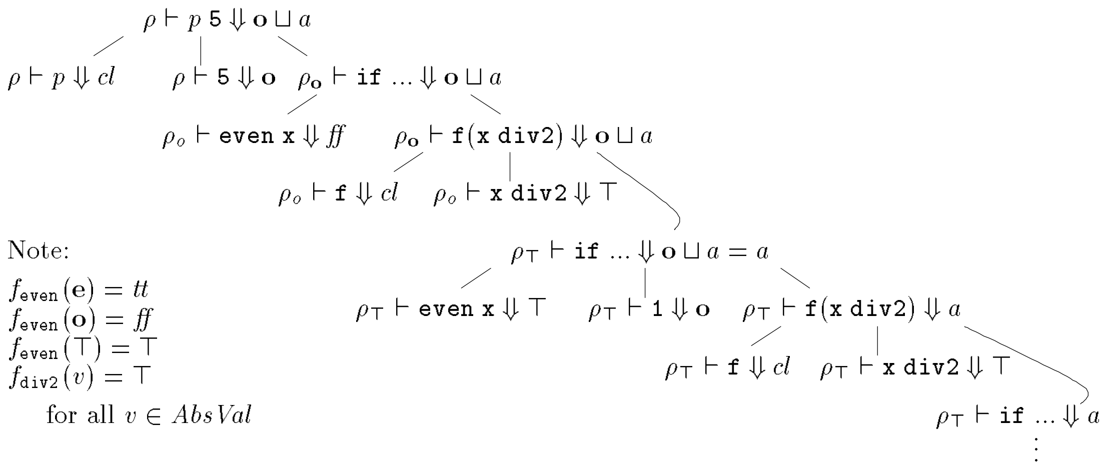


Figure 9: Abstraction interpretation of derivation

The safety relation on sequents is $(\rho_C \vdash e \Downarrow c)\ \mathit{safe}_{Seq}\ (\rho_A \vdash e \Downarrow a)$ iff $\rho_C\ \mathit{safe}_{Env}\ \rho_A$ and $c\ \mathit{safe}_{Val}\ a$. As before, a *c.i.*, t_C, is safely simulated by an *a.i.*, t_A, if $t_C\ \mathit{safe}_{Tree}\ t_A$ holds, where safe_{Tree} is the least relation such that $t_C\ \mathit{safe}_{Tree}\ t_A$ iff $\mathit{root}(t_C)\ \mathit{safe}_{Seq}\ \mathit{root}(t_A)$ and for every child subtree t_i of t_C, there exists a child subtree t_j of t_A such that $t_i\ \mathit{safe}_{Tree}\ t_j$.[14] The intuition is that every subtree in t_C is safely approximated by some subtree in t_A.

We desire the general result that for every source language program, p, concrete environment, ρ_C, and abstract environment, ρ_A, $\rho_C\ \mathit{safe}_{Env}\ \rho_A$ implies that for every $t_C \in \mathit{wftree}_{\mathcal{C}}$ such that $\mathit{root}(t_C) = \rho_C \vdash p \Downarrow c$, for every $t_A \in \mathit{wftree}_{\mathcal{A}}$ such that $\mathit{root}(t_A) = \rho_A \vdash p \Downarrow a$, it is the case that $t_C\ \mathit{safe}_{Tree}\ t_A$. The proof comes easily by induction on the height of the concrete derivation tree; see [32, 63] for example proofs in this style.

5.2 Infinite Abstract Derivations

Every program with a concrete tree should possess an abstract tree, and Figure 9 makes clear that infinite abstract derivations are necessary. This implies that the abstract semantics rule set, $\mathcal{A}$, defines by coinduction $\mathit{wftree}_{\mathcal{A}}$, which includes infinite well-formed derivations. Unfortunately, because of the synthesized attributes in the sequents, the coinductively defined set also includes multiple derivations for program p and its initial ρ_A—an example appears in Figure 9, where fixing $a = \mathbf{o}$ yields a well-formed infinite derivation, as does setting $a = \top$. For best precision one desires the least tree, which means one must partially order the set of derivation trees.[15]

The infinite trees do not impact safety: although the predicate safe_{Tree} is defined coinductively, the safety proof proceeds again by induction on the height of the concrete

[14] Note that j need not equal i, e.g., consider the *c.i.* and *a.i.* for `if e1 e2 e3`.
[15] This requires that the inference rules are monotone with respect to the ordering.

$$\frac{\rho \vdash e \Uparrow}{\rho \vdash op(e) \Uparrow} \qquad \frac{\rho \vdash e_1 \Uparrow}{\rho \vdash \texttt{if}\ e_1\ e_2\ e_3 \Uparrow}$$

$$\frac{\rho \vdash e_1 \Downarrow tt \quad \rho \vdash e_2 \Uparrow}{\rho \vdash \texttt{if}\ e_1\ e_2\ e_3 \Uparrow} \qquad \frac{\rho \vdash e_1 \Downarrow f\!\!f \quad \rho \vdash e_3 \Uparrow}{\rho \vdash \texttt{if}\ e_1\ e_2\ e_3 \Uparrow}$$

$$\frac{\rho \vdash e_1 \Uparrow}{\rho \vdash e_1\ e_2 \Uparrow} \qquad \frac{\rho \vdash e_1 \Downarrow \langle \rho', f', x', e' \rangle \quad \rho \vdash e_2 \Uparrow}{\rho \vdash e_1\ e_2 \Uparrow}$$

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

Figure 10: Negative big-step inference rules

tree, which remains finite. This works because any infinite paths in the abstract tree explore divergent computations that do not arise in the concrete tree.

5.3 Infinite Concrete Derivations

An inductive definition of the concrete semantics means that divergent programs cannot be studied. The obvious remedy is to use a coinductive interpretation, but the price one pays is that a divergent program, p, with its initial environment, ρ_C, might have multiple, well-formed, infinite derivations; a simple example comes from the rule scheme, $\frac{\rho \vdash \texttt{loop} \Downarrow v}{\rho \vdash \texttt{loop} \Downarrow v}$, which generates a family of distinct, well-formed infinite trees with roots of the form, $\rho_C \vdash \texttt{loop} \Downarrow v$, for all $v \in \mathit{Val}$. We would desire the least such tree, whose root is (apparently) $\rho_C \vdash \texttt{loop} \Downarrow \bot$, and indeed this approach can be formalized with the usual least fixed-point theory [70]. But the nonleast trees remain and they complicate the safety proofs. For this reason, we pursue an approach suggested by P. Cousot based on G∞-SOS [20] where there are two forms of inference rules, *positive* ones and *negative* ones: *(i)* the existing inference rules are the positive rules, and they generate finite, convergent derivation trees; *(ii)* new inference rules, the negative rules, are written specifically for divergent computation and are interpreted coinductively.

To formalize this, say that a sequent, $\rho \vdash e \Downarrow v$, for $v \in \mathit{Val}$, is *positive*; next, introduce a new sequent form, $\rho \vdash e \Uparrow$, representing divergence, and say that it is *negative*.

The concrete semantics has two rule sets:

- R^+, the *positive rules*, which are inference rules of the form $\frac{s_1 \cdots s_n}{s_0}$, where all sequents, s_i, $i \in 0..n$, are positive;
- R^-, the *negative rules*, which are inference rules of the form $\frac{s_1 \cdots s_n}{s_0}$, where s_0 is negative.

For the example, the positive rules are exactly the ones in Figure 6; Figure 10 shows the negative rules. For example, the first rule in Figure 10 states if e has a divergent derivation then so does $op(e)$.

The positive rules define the set of positive trees:

$$\begin{aligned}&\mathit{wftree}^+_C(t) \text{ iff there exists } \frac{s_1,\cdots,s_n}{\mathit{root}(t)} \in R^+ \text{ and for all child subtrees,}\\ &t_i, i \in 1\ldots n, \text{ of } t, \mathit{root}(t_i) = s_i \text{ and } \mathit{wftree}^+_C(t_i)\end{aligned}$$

Take the inductive interpretation of this predicate. Next, define the negative trees by

$$\begin{aligned}&\mathit{wftree}^-_C(t) \text{ iff there exists } \frac{s_1,\cdots,s_n}{\mathit{root}(t)} \in R^- \text{ and for all child subtrees,}\\ &t_i, i \in 1\ldots n, \text{ of } t, \mathit{root}(t_i) = s_i; \mathit{wftree}^+_C(t_i), 1 \le i < n, \text{ and } \mathit{wftree}^-_C(t_n)\end{aligned}$$

Take the coinductive interpretation of this predicate. The set of well-formed derivation trees is $\mathit{wftree}_C = \mathit{wftree}^+_C \cup \mathit{wftree}^-_C$. It is trivially true that $\mathit{wftree}^+_C \cap \mathit{wftree}^-_C = \{\}$.

The safety proof requires an extension to the definition of safe_{Seq} :

- $(\rho_C \vdash e \Downarrow c)\ \mathit{safe}_{Seq}\ (\rho_A \vdash e \Downarrow a)$ iff $\rho_C\ \mathit{safe}_{Env}\ \rho_A$ and $c\ \mathit{safe}_{Value}\ a$
- $(\rho_C \vdash e \Uparrow)\ \mathit{safe}_{Seq}\ (\rho_A \vdash e \Downarrow a)$ iff $\rho_C\ \mathit{safe}_{Env}\ \rho_A$

The definition of safe_{Tree} remains as before: $\mathit{safe}_{Tree} = \mathit{gfp}\, F$, where $F(S) = \{(t_C, t_A) \mid \mathit{root}(t_C)\ \mathit{safe}_{Seq}\ \mathit{root}(t_A)$ and for every child subtree t_i of t_C, there exists a child subtree t_j of t_A such that $(t_i, t_j) \in S\}$. As before, the goal is showing that every concrete tree starting from ρ_C, e is safely approximated by every abstract tree starting from ρ_A, e. This follows from the proof that $S_0 \subseteq F(S_0)$, where $S_0 = \{(t, t') \mid t \in \mathit{wftree}_C, t' \in \mathit{wftree}_A, \mathit{root}(t') = \rho_A \vdash e \Downarrow a$, and $((\mathit{root}(t) = \rho_C \vdash e \Downarrow c, \rho_C\ \mathit{safe}_{Env}\ \rho_A, c\ \mathit{safe}_{Val}\ a)$ or $(\mathit{root}(t) = \rho_C \vdash e \Uparrow, \rho_C\ \mathit{safe}_{Env}\ \rho_A))\}$.

The proof goes as follows: Consider a pair, $(t, t') \in S_0$; If $t \in \mathit{wftree}^+_C$, then we appeal to the earlier safety result for finite trees. If $t \in \mathit{wftree}^-_C$, then we must prove: *(i)* $\mathit{root}(t)\ \mathit{safe}_{Seq}\ \mathit{root}(t')$—since $\mathit{root}(t)$ is a negative sequent, this is immediate from the definition of S_0; *(ii)* for every child subtree, t_i of t, there is some child subtree, t_j, of t', such that $(t_i, t'_j) \in S_0$. This property must be verified by inspection of the inference rules used to derive t and t', respectively; we consider one example case: Say that the root of t is derived by this rule for divergent function application:

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

Consider subtree, t_1, whose root is $\rho_C \vdash e_1 \Downarrow \langle \rho'_C, f', x', e'\rangle$. This is a positive sequent, so we appeal to the safety result for finite concrete trees to verify that t'_1 must be a subtree whose root is $\rho_A \vdash e_1 \Downarrow \langle \rho'_A, f', x', e'\rangle$, where $\rho'_C\ \mathit{safe}_{Env}\ \rho'_A$. Hence, $(t_1, t'_1) \in S_0$. Similarly, $\mathit{root}(t_2) = \rho_C \vdash e_2 \Downarrow c'$, a finite tree, implying that $\mathit{root}(t'_2) = \rho_A \vdash e_2 \Downarrow a'$, $c'\ \mathit{safe}_{Val}\ a'$, and therefore $(t_2, t'_2) \in S_0$. Finally, for $\mathit{root}(t_3) = \rho'_C \oplus \{f' \mapsto \langle \rho'_C, f', x', e'\rangle, x' \mapsto c'\} \vdash e' \Uparrow$, we deduce from our knowledge about $\mathit{root}(t'_1)$ and $\mathit{root}(t'_2)$ that $\mathit{root}(t'_3) = \rho'_A \oplus \{f' \mapsto \langle \rho'_A, f', x', e'\rangle, x' \mapsto a'\} \vdash e' \Downarrow a$, for some $a \in \mathit{AbsVal}$. This implies $(t_3, t'_3) \in S_0$.

It is crucial for the simplicity of the proof that the negative sequents free us from reasoning about abstract synthesized attributes, e.g., a in the very last case above (cf. [70]).

5.4 Termination

We require that an *a.i.* terminate, and if the *AbsVal* set is infinite, then memoization can be utilized. Memoization proceeds as seen earlier: An *a.i.* is generated in stages, starting from a root sequent, $t_0 = \rho_A \vdash p \Downarrow \bot$. (The $\bot$ means that the synthesized attribute is unknown.) At stage $i+1$, each leaf in t_i is expanded by an inference rule; if $\rho \vdash e \Downarrow \bot$ is a newly generated leaf, and there is an ancestor node that also mentions e, then leaf is revised to $\rho \sqcup \rho' \vdash e \Downarrow \bot$, where ρ' is found in the leaf's nearest ancestor of the form $\rho' \vdash e \Downarrow a$. This gives t_{i+1}. When a leaf produces synthesized information, it is propagated upwards in the tree. If the limit of the sequence of t_is is an infinite tree, then the occurrences of $\Downarrow \bot$ are read as $\Uparrow$ instead.[16]

Memoization guarantees production of a regular tree when the *AbsVal* set is partially ordered and has the finite-chain property. This follows because a natural semantics is *semicompositional*,[17] that is, all syntax phrases that appear within the *a.i.* are subphrases of the original source program, thus there are a finite number of them.[18]

5.5 Application: Set-Based Analyses

The $\top$-value and the $\sqcup$-operation that appear in the abstract semantics destroy precision; we prefer to say $f_{\mathtt{div2}}(e)$ equals $\{\mathbf{e}, \mathbf{o}\}$ rather than $\top$. Worse still, the even-odd analysis of the program `(if even(m div2) (rec f x.0) (rec g y.1))n` joins the closures for `rec f x.0` and `rec g y.1`, producing $\top$, for which there is no inference rule for function application. An artificial rule for function application can be invented, but it is worthwhile to investigate "set-based" analyses [34, 44], where the abstract semantics rules use synthesized attributes that are sets. Three of the modified rules from Figure 6 are

$$\frac{\rho \vdash e \Downarrow S}{\rho \vdash op(e) \Downarrow \{f_{opA}(a) \;\mid\; a \in S\}}$$

$$\frac{\rho \vdash e_1 \Downarrow S_1 \quad \rho \vdash e_2 \Downarrow S_2 \quad \begin{array}{l}\rho_i + (f_i \mapsto cl_i) + (x_i \mapsto v_j) \vdash e_i \Downarrow S_{ij} \\ \qquad \text{for every } cl_i = \langle \rho_i, f_i, x_i, e_i \rangle \in S_1, v_j \in S_2\end{array}}{\rho \vdash e_1\, e_2 \Downarrow \cup\{S_{ij}\}}$$

$$\frac{\rho \vdash e_1 \Downarrow S_1 \qquad \rho \vdash e_2 \Downarrow S_2,\ \text{if } tt \in S_1 (\text{else } S_2 = \emptyset) \qquad \rho \vdash e_3 \Downarrow S_3,\ \text{if } f\!f \in S_1 (\text{else } S_3 = \emptyset)}{\rho \vdash \mathtt{if\ e1\ e2\ e3} \Downarrow S_2 \cup S_3}$$

For example, the first rule states that the evaluation of e yields a set of values, S; the result of $op(e)$ must therefore be the set of all $f_{opA}(a)$, for all $a \in S$.

The *a.i.* of the example in Figure 9 is reworked in Figure 11; the precision of the trace increases yet safety is preserved. Notice also that the recursion, $a = \{\mathbf{o}\} \cup a$, can be solved in the complete lattice $\mathcal{P}(AbsVal)$ (the computational ordering [20]), giving $a = \{\mathbf{o}\}$—there is no need for $\top$ and no need for an approximation ordering upon *AbsVal*.

[16] A more precise but lower-level explanation of big-step abstract-tree computation is found in [38].

[17] This term is due to Neil Jones.

[18] In contrast, a substitution-based semantics might not have this property. For example, the substitution semantics rule for function application would be $\frac{e_1 \Downarrow rec\ f\ x.e \quad e_2 \Downarrow v \quad [rec\ f\ x.e/f][v/x]e \Downarrow v'}{e_1\, e_2 \Downarrow v'}$.

let $p =$ `(rec f x.if even x 1 f(x div2))`
let $cl = \langle \rho, \mathtt{f}, \mathtt{x}, \mathtt{if\ even\ x}\ ...\rangle$
let $\rho_i = [\mathtt{f} \mapsto cl, \mathtt{x} \mapsto i]\rho$ in

$$\rho \vdash p\ \mathtt{5} \Downarrow \{\mathrm{o}\} \cup a$$

$$\rho \vdash p \Downarrow \{cl\} \qquad \rho \vdash \mathtt{5} \Downarrow \{\mathrm{o}\} \qquad \rho_{\mathrm{o}} \vdash \mathtt{if}\ ... \Downarrow \{\mathrm{o}\} \cup a = a$$

$$\rho_{\mathrm{o}} \vdash \mathtt{even\ x} \Downarrow \{\mathit{ff}\} \qquad \rho_{\mathrm{o}} \vdash \mathtt{f(x\ div2)} \Downarrow \{\mathrm{o}\} \cup a$$

$$\rho_{\mathrm{o}} \vdash \mathtt{f} \Downarrow \{cl\} \qquad \rho_{\mathrm{o}} \vdash \mathtt{x\ div2} \Downarrow \{\mathrm{e}, \mathrm{o}\}$$

$$\rho_{\mathrm{e}} \vdash \mathtt{if}\ ... \Downarrow \{\mathrm{o}\} \qquad \rho_{\mathrm{o}} \vdash \mathtt{if}\ ... \Downarrow a$$

$$\rho_{\mathrm{e}} \vdash \mathtt{even\ x} \Downarrow \{tt\} \qquad \rho_{\top} \vdash \mathtt{1} \Downarrow \{\mathrm{o}\}$$

Note:
$f_{\mathtt{even}}(\mathrm{e}) = tt$
$f_{\mathtt{even}}(\mathrm{o}) = \mathit{ff}$
$f_{\mathtt{div2}}(\mathrm{e}) = \{\mathrm{e}, \mathrm{o}\}$
$f_{\mathtt{div2}}(\mathrm{o}) = \{\mathrm{e}, \mathrm{o}\}$

Figure 11: Set-based abstract interpretation

When sets of closures appear in an *a.i.*, the analysis is called a *closure analysis* [39, 40, 62, 65, 73, 74]. The computational complexity of a closure analysis is high, and a major challenge is finding efficient, safe simulations.

To simplify notation, we represent a closure by a $\langle \rho, \ell \rangle$ pair, where ℓ is a unique "label" of a `rec` phrase, $\ell : \mathtt{rec\ f\ x.e}$. Next, we ignore primitive values and define

$$\begin{aligned} AbsVal &= \{\bullet\} \cup AbsClos \\ AbsClos &= \mathcal{P}(AbsEnv \times Label) \\ AbsEnv &= Id \stackrel{fin}{\to} AbsVal \end{aligned}$$

To reduce the overhead with sets, we redefine $AbsClos$ in an isomorphic form:

$$\begin{aligned} AbsVal &= \{\bullet\} \cup AbsClos \\ AbsClos &= Label \stackrel{fin}{\to} \mathcal{P}(AbsEnv) \\ AbsEnv &= Id \stackrel{fin}{\to} AbsVal \end{aligned}$$

Now, the task is finding simplifications of $AbsClos$. One that is related to the n-CFA analyses proposed by Shivers [62, 74] defines $AbsClos$ as $AbsClos_n$, for some fixed $n \geq 0$, where

$$\begin{aligned} AbsClos_n &= Label \stackrel{fin}{\to} \mathcal{P}(AbsEnv_n) \\ AbsEnv_0 &= \{\bullet\} \\ AbsEnv_{i+1} &= Id \stackrel{fin}{\to} AbsVal_i \\ AbsVal_i &= \{\bullet\} \cup AbsClos_i \end{aligned}$$

The set $AbsClos_n$ limits the depth to n of the closures that can be produced by the analysis, ensuring that the $AbsVal$ set is finite. Surprisingly useful analyses can be performed for $n = 0$ [73, 74], but a complication is that a function application must synthesize an environment for the closure $(\ell \mapsto \{\bullet\})$ when such a closure is applied to an argument. There are a variety of safe methods for synthesizing the environment—a natural one examines the derivation

$$e \ ::= \ 0 \ \mid \ \alpha.e \ \mid \ e_1 \,||\, e_2 \ \mid \ \mu p(e) \ \mid \ p$$

$$\alpha.e \stackrel{\alpha}{\to} e \qquad \frac{e_1 \stackrel{\alpha}{\to} e_1'}{e_1 \,||\, e_2 \stackrel{\alpha}{\to} e_1' \,||\, e_2} \qquad \frac{e \stackrel{\alpha}{\to} e'}{\mu p(e) \stackrel{\alpha}{\to} [\mu p(e)/p]e'} \qquad etc.$$

plus usual symmetric monoidal laws: $0 \,||\, e \equiv e$, $e_1 \,||\, e_2 \equiv e_2 \,||\, e_1$, etc.

Figure 12: Subset of CCS-like language

tree for sequents of the form $\rho' \vdash \ell : \texttt{rec f x.e} \Downarrow \{\ell \mapsto \{\bullet\}\}$ and joins the respective ρ's, thus (safely) confusing the creation sites (cf. "call sites" [75]) of the closure.

6 Small-step semantics

A third form of computation-tree-based semantics is small-step semantics, so called because it rewrites a program configuration, step by step, to a final or normal form. Here, we examine Plotkin-style structural operational semantics (*SOS*) [63, 67], where transitions are defined by inference rules. SOS semantics definitions include flowchart semantics, term-rewriting systems, and reduction systems.

This section shows that trace-based *a.i.* adapts nicely to small-step semantics, but a new problem arises because SOS definitions can generate new program syntax on the fly—arbitrary SOS derivations are not semicompositional and this can hinder the termination of an *a.i.* In response, we abstract upon the syntax of the source language itself to ensure termination of the *a.i.* We use as our running example Figure 12, which displays a nondeterministic CCS-like notation [53] with a small-step semantics that spawns processes via unfolding of a recursion constructor, μ.[19] Because the μ-rule generates new program syntax, the semantics derivations will not be semicompositional.

Figure 13 displays a simple example that shows how the operational semantics of the program $\mu x(\alpha.(\beta.x \,||\, x))$ generates new processes, that is, new program syntax. All paths in the behavior tree are infinite, and the newly generated processes make impossible a regular tree representation. To undertake a terminating abstract interpretation, some means must be found to limit the dynamically generated processes.

Since there are no semantic value sets in the example to be abstractly interpreted, an *a.i.* must *abstract the source program syntax itself.* To set the stage, we note that every program configuration has an isomorphic representation as a bag of processes, called a "process pool" [60, 61]. For example, the configuration $\beta.P \,||\, \beta.P \,||\, P$ is written as the bag $\{\beta.P, \beta.P, P\}$.

Next, as is common to abstract-interpretation applications, assume that the subphrases of the source program are indexed by "labels," ℓ_i, e.g.,

$$P = \ell_0\!: \mu x\ (\ell_1\!: \alpha.\ell_2\!: (\ell_3\!: \beta.\ \ell_4\!: x \,||\, \ell_5\!: x))$$

This means a process pool is just a function of the form $Label \to Nat$, e.g., the bag above

[19] The rule for μ in the figure unfolds a μ-process exactly once when it makes a communication step. This differs from the rule often used: $\dfrac{[\mu p(e)/p]e \stackrel{\alpha}{\to} e'}{\mu p(e) \stackrel{\alpha}{\to} e'}$, which operates on closed terms only but allows unbounded unfolding. Nonetheless, the rules generate bisimular computation trees.

Let $P = \mu x(\alpha.(\beta.x \mid\mid x))$ in

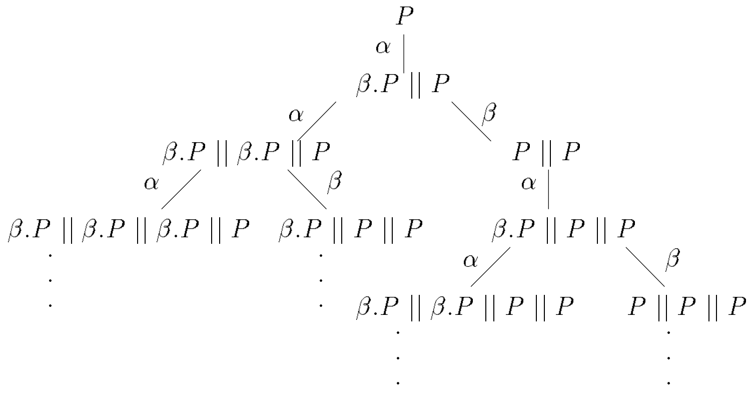


Figure 13: Example of process creation

Let $P = \mu x(\alpha.(\beta.x \mid\mid x))$ in

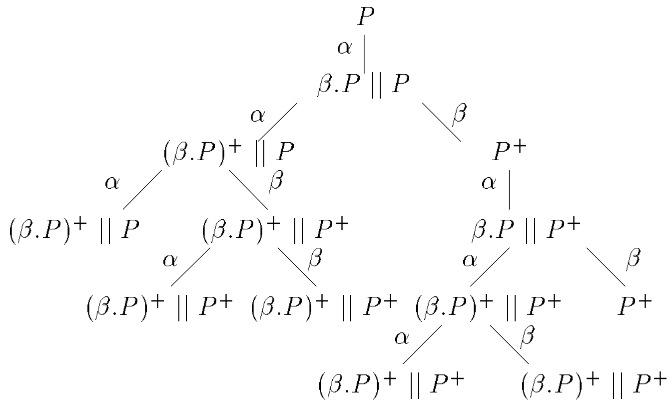


Figure 14: Abstract interpretation of process creation

is encoded by the function $[\ell_3 \mapsto 2][\ell_0 \mapsto 1](\lambda\ell.0)$.[20]

Our task is to abstractly represent the process pools, which we do by defining the abstract pools to be functions of the form $\mathit{Label} \to \{0, 1, \omega\}$ [13, 35]. Both domain and codomain of this function space are finite, so there exist a finite number of abstract pools. The example above is abstracted by $[\ell_3 \mapsto \omega][\ell_0 \mapsto 1](\lambda\ell.0)$, and the safety relation between concrete and abstract process pools is of course: for $cp \in \mathit{Pool}$, $ap \in \mathit{AbsPool}$,

$$cp\ \mathit{safe}_{Pool}\ ap \text{ iff for all } \ell \in \mathit{Label}, cp(\ell) \leq ap(\ell)$$

Note that parallel composition, $cp_1 \mid\mid cp_2$, is understood as bag union on process pools and its abstraction is $ap_1 \uplus ap_2 = \lambda\ell.(ap_1(\ell) \oplus ap_2(\ell))$, where $1 \oplus 1 = \omega$, and $m \oplus n = \max\{m, n\}$ otherwise.

Figure 14 shows the abstract interpretation of the derivation in Figure 13. For simplicity, an abstract pool is written as a regular expression, where a "+" marks a process whose

[20] In the example, ℓ_3 labels $\beta.x$ rather than $\beta.P$. The discrepency is due to the substitution semantics of μ-process unfolding. This is tolerated for now—treat x and P as "the same"—but will be repaired in the next section with an environment semantics.

count is ω. Using this representation of the abstract pools, we see that the semantics rules in Figure 12 can be used as the abstract semantics rules along with one new rule which accounts for processes whose count is ω:

$$\frac{e \stackrel{\alpha}{\to} e'}{e^{+} \stackrel{\alpha}{\to} e^{+} \,||\, e'}$$

Since there are a finite number of abstract pools, that is, syntax configurations, the *a.i.* must be a regular tree.

The purpose of abstractly interpreting the syntax is to restore a form of semicompositionality. The regular expressions used here were first devised by Codish, Falaschi and Marriott as "*-abstractions" [12]. In general, one might need a context-free grammar representation of syntax configurations to recover semicompositionality; the precedent here is due to Giannotti and Latella [28]; see also [7, 37].

6.1 Safety

Safety is stated as before—the *a.i.* is a simulation of the *c.i.* :

$$\begin{array}{l} t\ \mathit{safe}_{Trace}\ u \text{ iff } \mathit{root}(t)\ \mathit{safe}_{Pool}\ \mathit{root}(u) \\ \quad \text{and for every transition, } t \stackrel{\alpha}{\longrightarrow} t_i, \text{ there exists a transition,} \\ \quad u \stackrel{\alpha}{\longrightarrow} u_j, \text{ such that } t_i\ \mathit{safe}_{Trace}\ u_j \end{array}$$

The safety result follows from this relational homomorphic property of the semantics rules:

$$\begin{array}{c} cp\ \mathit{safe}_{Pool}\ ap \text{ and } cp \stackrel{\alpha}{\to} cp' \text{ imply there exists } ap' \\ \text{such that } ap \stackrel{\alpha}{\to} ap' \text{ and } cp'\ \mathit{safe}_{Pool}\ ap' \end{array}$$

6.2 Application: Abstraction on Syntax and Semantics

Now that abstraction of syntax is understood, we consider a full-blown example, where we must abstract upon both program syntax and input data. Our example is CCS with value passing, where the values are channel names. The relevant semantics rules are

$$\alpha?x.e \stackrel{\alpha?\alpha'}{\to} [\alpha'/x]e \qquad \alpha!\alpha'.e \stackrel{\alpha!\alpha'}{\to} e \qquad \frac{e_1 \stackrel{\alpha?\alpha'}{\to} e_1' \quad e_2 \stackrel{\alpha!\alpha'}{\to} e_2'}{e_1 \,||\, e_2 \stackrel{\tau}{\to} e_1' \,||\, e_2'}$$

where τ represents an internal step, a "tau move."

This semantics defines argument transmission via substitution, but substitutions generate new program syntax, which hampers termination of an abstract interpretation. For this reason, we revise the semantics into an environment semantics: each process, e, owns a local environment, ρ, that holds bindings of identifiers to channels, and we write a process configuration as $\rho \vdash e$. To simplify matters, assume that argument identifiers are distinct from process indentifiers; also, assume that each recursive process, $\mu p(e)$, uses a unique process identifier, p. Although it is not essential, we eliminate the substitution semantics for $\mu p(e)$ by saving a "closure," $\{p \mapsto \rho\}$, of $\mu p(e)$ in the environment of the unfolded process, e. (Remember that the process id, p, uniquely identifies the code, $\mu p(e)$.) Environments hold bindings of argument identifiers to channels, $\{x_i = \alpha_i\}$, and bindings of process identifiers to "closures," $\{p_j \mapsto \rho_j\}$. See Figure 15 for the results.

$c \in$ Expression-configuration $\quad p \in$ Process-identifier
$e \in$ Expression $\quad \rho \in$ Environment
$g \in$ Guard $\quad v \in$ Channel-expression
$x \in$ Argument-identifier $\quad \alpha \in$ Channel
$\Delta \in$ Transition-label

$$\begin{array}{lcl} c & ::= & \rho \vdash e \;\mid\; c_1 \,||\, c_2 \\ e & ::= & 0 \;\mid\; g.e \;\mid\; e_1 \,||\, e_2 \;\mid\; \mu p(e) \;\mid\; p \\ g & ::= & v?x \;\mid\; v_1!v_2 \\ v & ::= & \alpha \;\mid\; x \\ \rho & ::= & \{x_i = \alpha_i\} \cup \{p_j \mapsto \rho_j\} \\ \Delta & ::= & \tau \;\mid\; \alpha_1?\alpha_2 \;\mid\; \alpha_1!\alpha_2 \end{array}$$

Note: assume each recursive process, $\mu p(e)$, uses a unique process identifier, p.
Congruences: usual symmetric, monoidal rules (e.g., $\rho \vdash 0 \,||\, e \equiv \rho \vdash e$), plus:

$$\rho \vdash (e_1 \,||\, e_2) \equiv (\rho \vdash e_1) \,||\, (\rho \vdash e_2) \qquad \frac{(x = \alpha) \in \rho}{\rho \vdash g.e \equiv \rho \vdash ([\alpha/x]g).e} \qquad \frac{(p \mapsto \rho) \in \rho'}{\rho' \vdash p \equiv \rho \vdash \mu p(e)}$$

Computation rules:

$$\rho \vdash \alpha?x.e \xrightarrow{\alpha?\alpha'} \rho \oplus \{x = \alpha'\} \vdash e \qquad \rho \vdash \alpha!\alpha'.e \xrightarrow{\alpha!\alpha'} \rho \vdash e$$

$$\frac{\rho \vdash e \xrightarrow{\Delta} \rho' \vdash e'}{\rho \vdash \mu p(e) \xrightarrow{\Delta} \rho' \oplus \{p \mapsto \rho\} \vdash e'} \qquad \frac{c_1 \xrightarrow{\alpha?\alpha'} c_1' \quad c_2 \xrightarrow{\alpha!\alpha'} c_2'}{c_1 \,||\, c_2 \xrightarrow{\tau} c_1' \,||\, c_2'} \qquad \frac{c_1 \xrightarrow{\Delta} c_1'}{c_1 \,||\, c_2 \xrightarrow{\Delta} c_1' \,||\, c_2}$$

Figure 15: Concrete small-step semantics of channel passing

The computation rules are kept simple by employing the usual symmetric, monoidal congruences plus three more: the first additional congruence explains how an environment is copied to component processes of a system; the second shows how identifier lookup in the environment is employed; and the third shows that lookup of a recursive process identifier, p, causes p to be replaced by the code, $\mu p(e)$, and the environment, ρ, that it names. The computation rules are as expected; the only novel rule is the one for recursive processes: it creates a closure, $\{p \mapsto \rho\}$, when the recursively defined process, $\rho \vdash \mu p(e)$, is unfolded. Figure 16 displays an example concrete tree, where each τ-transition is annotated by the channel interaction that caused it.

Now, a program's process pool representation is a bag of (environment,process) pairs. Assuming that processes are labelled, process pools have the form $Pool = Label \rightarrow Bag(Environment)$, where $Environment = (ArgumentId \rightarrow Channel) \times (ProcessId \rightarrow Environment)$.[21] (Recall that an environment holds bindings, $\{x = \alpha\}$, of identifiers to channels and also bindings, $\{p \mapsto \rho\}$—closures—of process identifiers to the environments used by the processes.)

A manageable abstract semantics defines an abstract pool to be a function in $AbsPool = Label \rightarrow AbsEnv \times \{0, 1, \omega\}$, where abstract environments are defined $AbsEnv =$

[21] We work only with well-founded, that is, inductively defined, environments.

Let $P = \mu p(\alpha?x.(x!x.0 \,||\, p))$ and $\rho_a = \{x = a, p \mapsto \emptyset\}$ in

$$\emptyset \vdash \alpha!\alpha.0 \,||\, \emptyset \vdash \alpha!\beta.0 \,||\, \emptyset \vdash P$$

$\tau(\alpha!\alpha)$ (left) ; $\tau(\alpha!\beta)$ (right): (dual to left subtree)

$$\emptyset \vdash \alpha!\beta.0 \,||\, \rho_\alpha \vdash x!x.0 \,||\, \rho_\alpha \vdash p$$
$$\equiv \emptyset \vdash \alpha!\beta.0 \,||\, \rho_\alpha \vdash \alpha!\alpha.0 \,||\, \emptyset \vdash P$$

$\tau(\alpha!\alpha)$ (left):

$$\emptyset \vdash \alpha!\beta.0 \,||\, \rho_\alpha \vdash x!x.0 \,||\, \rho_\alpha \vdash p$$
$$\equiv \emptyset \vdash \alpha!\beta.0 \,||\, \rho_\alpha \vdash \alpha!\alpha.0 \,||\, \emptyset \vdash P$$

$\tau(\alpha!\alpha)$ ⋮ $\quad$ $\tau(\alpha!\beta)$ ⋮

$\tau(\alpha!\beta)$ (right):

$$\rho_\alpha \vdash \alpha!\alpha.0 \,||\, \rho_\beta \vdash x!x.0 \,||\, \rho_\beta \vdash p$$
$$\equiv \rho_\alpha \vdash \alpha!\alpha.0 \,||\, \rho_\beta \vdash \beta!\beta.0 \,||\, \emptyset \vdash P$$

$\tau(\alpha!\alpha)$

$$\rho_\beta \vdash \beta!\beta.0 \,||\, \rho_\alpha \vdash x!x.0 \,||\, \rho_\alpha \vdash p$$
$$\equiv \rho_\beta \vdash \beta!\beta.0 \,||\, \rho_\alpha \vdash \alpha!\alpha.0 \,||\, \emptyset \vdash P$$

$\tau(\alpha!\alpha)$

⋮

Figure 16: Concrete interpretation of channel passing

$(ArgumentId \rightarrow \mathcal{P}(Channel)) \times (ProcessId \rightarrow AbsEnv)$. That is, an abstract environment associates an argument identifier with a set of possible channels that the identifier might denote and it associates a process identifier with the abstract environment used by the identifier's recursively defined process.

The safety relation between concrete and abstract process pools becomes: for all $cp \in Pool, ap \in AbsPool$,

$$\begin{array}{c} cp \; safe_{Pool} \; ap \text{ iff } \text{ for all } \ell \in Label, |cp(\ell)| \leq (ap(\ell) \downarrow 2) \\ \text{and for all } \rho \in cp(\ell), \rho \; safe_{Env} \; (ap(\ell) \downarrow 1) \end{array}$$

where concrete and abstract environments are related as follows:

$$\begin{array}{l} (\{x_i = \alpha_i\}, \{p_j \mapsto \rho_j\}) \; safe_{Env} \; (\{x_i = S_i\}, \{p_j \mapsto \rho'_j\}) \\ \text{iff } (i) \text{ for all } x_i,\ \alpha_i \in S_i, \text{ and } (ii) \text{ for all } p_j,\ \rho_j \; safe_{Env} \; \rho'_j \end{array}$$

Based on the above definitions, it is easy to see, for example, that for concrete pool $\{x = \alpha\} \vdash x!x.0 \,||\, \{x = \beta\} \vdash x!x.0 \,||\, \emptyset \vdash \alpha?y.0$ the abstract pool that best describes it is $\{x = \{\alpha, \beta\}\} \vdash (x!x.0)^+ \,||\, \emptyset \vdash \alpha?y.0$.

As in the previous section, a new computation rule is needed for the abstract pools:

$$\frac{\rho \vdash e \overset{\Delta}{\rightarrow} \rho' \vdash e'}{\rho \vdash e^+ \overset{\Delta}{\rightarrow} \rho \vdash e^+ \,||\, \rho' \vdash e'}$$

If we use the congruence rule in Figure 15 to define substitution in the abstract semantics, then a set of channels is substituted for a channel identifier, e.g., $\{x = \{\alpha, \beta\}\} \vdash x!x.0 \equiv$

$\{x = \{\alpha, \beta\}\} \vdash \{\alpha, \beta\}!\{\alpha, \beta\}.0$. This implies that we should use the following safe, simple, but inexact abstract rules for communication:

$$\rho \vdash S?x.e \stackrel{\alpha?S'}{\rightarrow} \rho \oplus \{x = S'\} \vdash e \ \text{ for } \alpha \in S \qquad\qquad \rho \vdash S!S'.e \stackrel{\alpha!S'}{\rightarrow} \rho \vdash e \ \text{ for } \alpha \in S$$

This causes the entire set of channels, S', to be transmitted along any $\alpha \in S$. This is a form of "independent attribute" analysis of channel flow [43].

In contrast, a "relational analysis" would keep distinct the channels in S; we can formalize this idea with the following congruence rule:

$$\frac{(x = S) \in \rho}{\rho \vdash g.e \equiv \sum_{\alpha \in S} \rho \vdash ([\alpha/x]g).e}$$

The substitution of the elements of a set, S, into $g.e$ generates not one but a set of new abstract processes to choose from. For example, $\{x = \{\alpha, \beta\}\} \vdash x!x.0 \equiv (\{x = \{\alpha, \beta\}\} \vdash \alpha!\alpha.0) + (\{x = \{\alpha, \beta\}\} \vdash \beta!\beta.0)$.[22] But more importantly, we have, if $(x = S) \in \rho$, then $\rho \vdash (g.e)^{+}$ is bisimilar to $||_{\alpha \in S} \ (\rho \vdash (([\alpha/x]g).e)^{+})$. This result ensures that a process configuration can be understood still as a process pool.[23]

Of course, the price paid for the extra precision of the relational analysis is a slower convergence of construction of the regular tree.

Figure 17 shows the regular tree that results from the relational analysis of the program in Figure 16. The interesting stages in the analysis are lettered (a)-(f). The transition into node (a), that is, the transmission of $\alpha!\beta$, generates the configuration $\rho_\alpha \vdash x!x.0 \ || \ \rho_\beta \vdash x!x.0 \ || \ \rho_\beta \vdash p$, which has as its abstract representation node (a). The equivalence for process identifier lookup gives node (b); and the equivalence for argument identifier lookup gives (c), which is parenthesized to emphasize that node (c) is not actually generated—the analysis does not generate new source program syntax. (As just stated, $\rho_{\{\alpha,\beta\}} \vdash (x!x.0)^{+}$ is bisimilar to $\rho_{\{\alpha,\beta\}} \vdash (\alpha!\alpha.0)^{+} \ || \ \rho_{\{\alpha,\beta\}} \vdash (\beta!\beta.0)^{+}$.) The next transition, caused by a τ-step due to $\alpha!\alpha$, generates node (d), which is a syntax configuration that appeared earlier at (a); so, node (e) is a result of the memoization process, which joins the respective environments of (a) and (d). The substitution for p uncovers node (f), which is a repetition of (b).

7 Conclusion

We have demonstrated how the techniques of trace-based abstract interpretation can yield simple and systematic *a.i.* s for computation-tree-based operational semantics definitions. As noted earlier, a fundamental advantage of trace-based *a.i.* is its simple formulation of collecting semantics extraction; this arises because trace-based *a.i.* uses semantic structures that are regular trees, that is, finite-state graphs.

Of course, model-checking techniques operate on finite-state graphs, and it is an obvious step to apply model checking to the abstract trees generated from a trace-based *a.i.* In particular, the trace-based *a.i.* methodology makes clear that the objective of model checking is to calculate the collecting semantics of a graph [72]. This insight opens the door to a

[22] The infix "+" operator is external choice, and the usual computation rules for it would be required. We will not develop this concept further in this paper.

[23] Recall that two processes are *bisimular* if they simulate each other [53]. This implies that the trace trees of the two processes are interchangeable for our purposes.

Let $P = \mu p(\alpha?x.(x!x.0 \,||\, p))$ and $\rho_S = \{x = S, p \mapsto \emptyset\}$ in

$$\emptyset \vdash \alpha!\alpha.0 \,||\, \emptyset \vdash \alpha!\beta.0 \,||\, \emptyset \vdash P$$

$\tau(\alpha!\alpha)$ $\qquad$ $\tau(\alpha!\beta)$

$\emptyset \vdash \alpha!\beta.0 \,||\, \rho_\alpha \vdash x!x.0 \,||\, \rho_\alpha \vdash p$ $\qquad$ (dual to left subtree)

$\equiv \emptyset \vdash \alpha!\beta.0 \,||\, \rho_\alpha \vdash x!x.0 \,||\, \emptyset \vdash P$

$(\equiv \emptyset \vdash \alpha!\beta.0 \,||\, \rho_\alpha \vdash \alpha!\alpha.0 \,||\, \emptyset \vdash P)$

$\tau(\alpha!\alpha)$ $\qquad$ $\tau(\alpha!\beta)$

$\emptyset \vdash \alpha!\beta.0 \,||\, \rho_\alpha \vdash x!x.0 \,||\, \rho_\alpha \vdash p$ $\qquad$ (a) $\rho_{\{\alpha,\beta\}} \vdash (x!x.0)^+ \,||\, \rho_\beta \vdash p$

(b) $\equiv \rho_{\{\alpha,\beta\}} \vdash (x!x.0)^+ \,||\, \emptyset \vdash P$

(c) $(\equiv \rho_{\{\alpha,\beta\}} \vdash (\alpha!\alpha.0)^+ \,||\, \rho_{\{\alpha,\beta\}} \vdash (\beta!\beta.0)^+ \,||\, \emptyset \vdash P)$

$\tau(\alpha!\alpha)$

(d) $\rho_{\{\alpha,\beta\}} \vdash (x!x.0)^+ \,||\, \rho_\alpha \vdash p$

(e) $\rho_{\{\alpha,\beta\}} \vdash (x!x.0)^+ \,||\, \rho_{\{\alpha,\beta\}} \vdash p$

(f) $\equiv \rho_{\{\alpha,\beta\}} \vdash (x!x.0)^+ \,||\, \emptyset \vdash P$

Figure 17: Abstract interpretation of channel passing

potentially long and fruitful interaction between the fields of *a.i.* and model checking, of which initial examples are [10, 26, 49, 78, 80].

8 Acknowledgements

Mitchell Wand studied an earlier draft of this paper and made many useful suggestions. Helpful comments and assistance were provided by Anindya Banerjee, Ed Clarke, Radhia and Patrick Cousot, Olivier Danvy, Chris Hankin, Husain Ibraheem, Daniel Jackson, Thomas Jensen, Daniel LeMétayer, Hanne and Flemming Nielson, Frank Pfenning, David Sands, Colin Stirling, Andrew Wright, and the paper's anonymous referees.

References

[1] S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *J. of Logic and Computation*, 1:5–40, 1990.

[2] S. Abramsky and C. Hankin, editors. *Abstract interpretation of declarative languages.* Ellis Horwood, Chichester, 1987.

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

[4] A. Aiken and N. Heintze. Constraint-based program analysis. Technical Report `http://http.cs.berkeley.edu/~aiken/popl95.ps`, Tutorial talk at 22nd ACM Symp. Principles of Programming Languages, 1995.

[5] A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *21st ACM Symp. on Principles of Programming Languages*, Portland, Oregon, 1994.

[6] G. Bruns. A practical technique for process abstraction. In *4th International Conference on Concurrency Theory (CONCUR'93)*, Lecture Notes in Computer Science 715, pages 37–49. Springer-Verlag, 1993.

[7] O. Burkart and B. Steffen. Model checking for context-free processes. In *Proc. CONCUR92*, Lecture Notes in Computer Science 630, pages 123–137. Springer, 1992.

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

[9] E.M. Clarke, O. Grumberg, and D.E. Long. Verification tools for finite-state concurrent systems. In J.W. deBakker, W.-P. deRoever, and G. Rozenberg, editors, *A Decade of Concurrency: Reflections and Perspectives*, number 803 in Lecture Notes in Computer Science, pages 124–175. Springer, 1993.

[10] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[11] R. Cleaveland, P. Iyer, and D. Yankelevich. Optimality in abstractions of model checking. In *SAS'95: Proc. 2d. Static Analysis Symposium*, Lecture Notes in Computer Science 983, pages 51–63. Springer, 1995.

[12] M. Codish, M. Falaschi, and K. Marriott. Suspension analysis for concurrent logic programs. In *Proc. 8th Int'l. Conf. on Logic Programming*, pages 331–345. MIT Press, 1991.

[13] C. Colby. Analyzing the communication topology of concurrent programs. In *ACM Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'95)*, pages 202–214, 1995.

[14] C. Consel and S.C. Khoo. Parameterized partial evaluation. *ACM Trans. Prog. Lang. and Sys.*, 15(3):463–493, 1993.

[15] G. Cousineau and M. Nivat. On rational expressions representing infinite rational trees. In *8th Conf. Math. Foundations of Computer Science: MFCS'79*, Lecture Notes in Computer Science 74, pages 567–580. Springer, 1979.

[16] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes.* PhD thesis, University of Grenoble, 1978.

[17] 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.

[18] 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.

[19] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[20] 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.

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

[22] D. Dams. *Abstract interpretation and partition refinement for model checking*. PhD thesis, Technische Universiteit Eindhoven, The Netherlands, 1996.

[23] D. Dams, O. Grumberg, and R. Gerth. Abstract intepretation of reactive systems. In E.-R. Olderog, editor, *Proc. IFIP Working Conference on Programming Concepts, Methods, and Calculi.* North-Holland, 1994.

[24] Alain Deutsch. *Modeles Operationnels de Langage de Programmation et Representations de Relations sur des Langages Rationnels.* PhD thesis, University of Paris VI, 1992.

[25] 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.

[26] M. Dwyer and D. Schmidt. Limiting state explosion with filter-based refinement. In Annalisa Bossi, editor, *International Workshop on Verification, Model Checking and Abstract Interpretation, Port Jefferson, Long Island, N.Y.*, `http://www.cis.ksu.edu/~schmidt/papers/filter.ps.Z`, 1997.

[27] E.A. Emerson and C.L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *First Annual Symposium on Logic in Computer Science*, pages 267–278. IEEE, 1986.

[28] F. Giannotti and D. Latella. Gate splitting in LOTOS specifications using abstract interpretation. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93*, number 668 in Lecture Notes in Computer Science, pages 437–452. Springer-Verlag, 1993.

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

[30] 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.

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

[32] C. Gunter. *Semantics of Programming Languages.* MIT Press, Cambridge, MA, 1992.

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

[34] N. Heintze. Set-based analysis of ML programs. In *Proc. ACM Symp. Lisp and Functional Programming*, pages 306–317, 1994.

[35] P. Hudak. A semantic model of reference counting and its abstraction. In S. Abramsky and C. Hankin, editors, *Abstract interpretation of declarative languages*, pages 45–62. Ellis Horwood, Chichester, 1987.

[36] 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.

[37] H. Hungar and B. Steffen. Local model checking for context-free processes. In *Proc. ICALP93*, Lecture Notes in Computer Science 700, pages 593–605. Springer, 1993.

[38] H. Ibraheem and D. Schmidt. Adapting big-step semantics to small-step style. In C. Talcott, editor, *Proc. 2d Workshop on Higher-Order Techniques in Operational Semantics.* Elsevier Electronic Notes in Theoretical Computer Science, 1998.

[39] 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.

[40] S. Jagannathan and A. Wright. Effective flow analysis for avoiding run-time checks. In A. Mycroft, editor, *Static Analysis Symposium*, number 983 in Lecture Notes in Computer Science. Springer-Verlag, 1995.

[41] N. Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 4*, pages 527–636. Oxford Univ. Press, 1995.

[42] N.D. Jones. The essence of program transformation by partial evaluation and driving. In N.D. Jones, N. Hagiya, and S. Masahiko, editors, *Logic, Language, and Computation: a Festscrift in Honor of Satoru Takasu*, pages 206–224. Lecture Notes in Computer Science 792, Springer-Verlag, 1994.

[43] 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.

[44] 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.

[45] S. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In *FPCA'89: Functional Programming and Computer Architecture*, pages 54–74. ACM Press, 1989.

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

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

[48] K. Kennedy. A survey of data flow analysis techniques. In S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 5–54. Prentice-Hall, 1981.

[49] M. Klein, D. Koschuetzki, J. Knoop, and B. Steffen. DFA&OPT-MetaFrame: a tool kit for program analysis and optimization. In *Proc. TACAS'96*, pages 422–426. Lecture Notes in Computer Science 1055, Springer, Berlin, 1996.

[50] F. Levi. Abstract model checking of value-passing processes. In Annalisa Bossi, editor, *International Workshop on Verification, Model Checking and Abstra ct Interpretation, Port Jefferson, Long Island, N.Y.*, `http://www.dsi.unive.it/~bossi/VMCAI.html`, 1997.

[51] K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[52] 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.

[53] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

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

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

[56] 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.

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

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

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

[60] F. Nielson and H. R. Nielson. Higher-order concurrent programs with finite communication topology. In *Proc. ACM POPL'94*, pages 84–97, 1994.

[61] F. Nielson and H. R. Nielson. From CML to its process algebra. *Theoretical Computer Science*, 155(1):179–220, 1996.

[62] F. Nielson and H. R. Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proc. ACM POPL'97*, 1997.

[63] H. R. Nielson and F. Nielson. *Semantics with Applications, a formal introduction.* Wiley Professional Computing. John Wiley and Sons, 1992.

[64] 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.

[65] J. Palsberg. Closure analysis in constraint form. *ACM Trans. Programming Languages and Systems*, 17(1):47–62, 1995.

[66] D. Park. Concurrency and automata in infinite strings. Lecture Notes in Computer Science 104, pages 167–183. Springer, 1981.

[67] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[68] S. Purushothaman and J. Seaman. From operational semantics to abstract semantics. In *Proc. ACM Conf. Functional Programming and Computer Architecture.* ACM Press, 1993.

[69] D. Sands. Total correctness by local improvement in program transformation. In *Proc. 22nd Symp. on Principles of Prog. Languages*, pages 221–232. ACM Press, 1995.

[70] D.A. Schmidt. Natural-semantics-based abstract interpretation. In A. Mycroft, editor, *Static Analysis Symposium*, number 983 in Lecture Notes in Computer Science, pages 1–18. Springer-Verlag, 1995.

[71] D.A. Schmidt. Abstract interpretation of small-step semantics. In M. Dam and F. Orava, editors, *Proc. 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, Lecture Notes in Computer Science. Springer-Verlag, 1996.

[72] D.A. Schmidt. Data-flow analysis is model checking of abstract interpretations. In *Proc. 25th ACM Symp. on Principles of Prog. Languages.* ACM Press, 1998.

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

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

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

[76] B. Steffen. Generating data-flow analysis algorithms for modal specifications. *Science of Computer Programming*, 21:115–139, 1993.

[77] B. Steffen. Property-oriented expansion. In R. Cousot and D. Schmidt, editors, *Static Analysis Symposium: SAS'96*, volume 1145 of *Lecture Notes in Computer Science*, pages 22–41. Springer-Verlag, 1996.

[78] B. Steffen, A. Classen, M. Klein, J. Knoop, and T. Margaria. The fixpoint analysis machine. In I. Lee and S. Smolka, editors, *Proc. CONCUR'95*, volume 962 of *Lecture Notes in Computer Science*, pages 72–87. Springer-Verlag, 1995.

[79] C. Stirling. Modal and temporal logics. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 477–563. Oxford University Press, 1992.

[80] A. Venet. Abstract interpretation of the pi-calculus. In M. Dam and F. Orava, editors, *Proc. LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, Lecture Notes in Computer Science. Springer, 1996.

[81] M. Wand and P. Steckler. Selective and lightweight closure conversion. In *Proc. 21st Symp. on Principles of Prog. Languages*, pages 435–445. ACM Press, 1994.

[82] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Infomaticae*, 10:115–122, 1987.