Basic Research in Computer Science

BRICS RS-97-2 D. A. Schmidt: Abstract Interpretation in the Operational Semantics Hierarchy

Abstract Interpretation in the Operational Semantics Hierarchy

David A. Schmidt

BRICS Report Series **RS-97-2**

ISSN 0909-0878 **March 1997**

Copyright © 1997, BRICS, Department of Computer Science University of Aarhus. All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

See back inner page for a list of recent BRICS Report Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/97/2/`

Abstract Interpretation in the Operational Semantics Hierarchy

David A. Schmidt
BRICS*

March 18, 1997

Abstract

We systematically apply the principles of Cousot-Cousot-style abstract interpretation (*a.i.*) to the hierarchy of operational semantics definitions—flowchart, big-step, and small-step semantics. For each semantics format we examine the principles of safety and liveness interpretations, first-order and second-order analyses, and termination properties. Application of *a.i.* to data-flow analysis, model checking, closure analysis, and concurrency theory are demonstrated. Our primary contributions are separating the concerns of safety, termination, and efficiency of representation and showing how *a.i.* principles apply uniformly to the various levels of the operational semantics hierarchy and their applications.

*Basic Research in Computer Science, Centre of the Danish National Research Foundation. Permanent address: Computing and Information Sciences Department, Kansas State University, Manhattan, KS 66506 USA. schmidt@cis.ksu.edu. Also partially supported by NSF CCR-9302962 and CCR-9633388.

1 Introduction

Abstract interpretation (*a.i.*) is accepted as the correctness foundation for data-flow analysis of flowchart programs [11, 12, 31], and related research has demonstrated that *a.i.* can be applied to nonflowchart programs defined by denotational semantics [1, 6, 15, 20, 31, 35, 42, 51, 45, 46, 47] and structural operational semantics [13, 24, 56, 57, 58, 59, 66]. Model checking is another important applications area [8, 17, 18, 63, 64].

In this paper, we survey abstract interpretation in the hierarchy of operational semantics: flowchart semantics, big-step (natural) semantics, and small-step semantics. We define it, explain how to do it, show how to terminate it, and apply it to data-flow analysis, model checking, and concurrency theory. We examine the distinctions between safety and liveness interpretations and first-order and second-order analyses (collecting semantics), and we handle challenges that arise in the semantics forms: Big-step semantics cannot express divergence, so we employ coinductive definition techniques in response; small-step semantics generate sequences of program configurations that are unbounded in size, so we abstractly interpret source language syntax itself.

The paper's technical concepts are taken from the trailblazing research of Cousot and Cousot [16, 11, 12, 13, 14, 15]; our contribution is the expository and systematic use of these concepts in an important applications arena.

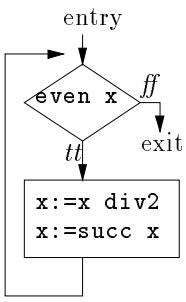
The structure of the paper goes as follows: Basic concepts appear in Section 1.1; Section 2 applies the concepts to a thorough development of abstract interpretation of flowchart semantics. Sections 3 and 4 apply *a.i.* to big-step semantics and small-step semantics, respectively, addressing problems unique to these formats. Applications are intertwined with the semantic forms upon which they are based. Section 5 concludes.

1.1 What is Abstract Interpretation?

Given that the *concrete interpretation* (*c.i.*) of a program is the execution trace of the program applied to run-time data, we say that the *abstract interpretation* (*a.i.*) is the execution trace of the program applied to tokens that denote properties of the run-time data—an *a.i.* is a "symbolic execution" where the symbols have semantic content. An example is 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.

When run-time data sets are replaced by tokens, the operations within the program must be revised to compute consistently on the tokens. In algebraic terminology, the program's flowchart is a "signature"; when the flowchart's boxes are instantiated with operations that compute on run-time data, one obtains a *c.i.* of the signature; when the boxes are instantiated with 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 the operation `y:=x+1` is the usual assignment, and the abstract semantics is a type inference: `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. For example, the abstract interpretation of a test, `x>0`, cannot be decided when the token value of `x` is *int*.



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

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

Figure 1: Flowchart and concrete interpretation

This forces the *a.i.* to traverse both execution paths that emanate from the test, implying that loop paths can be traversed forever. Therefore, an *a.i.* must be coupled with a strategy for termination. The strategy must ensure a program's *a.i.* is a trace where every infinite path contains a node that is a repetition of one seen earlier in the path, that is, the trace is a *regular tree*. Techniques like memoization [58, 59] and widening [11] can ensure regular trees.

Once an *a.i.* is terminated, one must extract information from it and apply the information to validation or code improvement. The information extracted is the *collecting semantics*; both *c.i.* and *a.i.* possess collecting semantics, which can be *first-* or *second-order* [45]. A first-order 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 the program points. A second-order collecting semantics maps program points to the set of execution paths that lead into (or, dually, lead out from) the program points. 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.*

The usual collecting semantics for a type inference is first order, whereas the collecting semantics for an available-expressions analysis is (forwards) second-order, and a live-variable analysis produces a (backwards) second-order collecting semantics. There exist more general forms of collecting semantics [13], which are discussed later.

For efficiency, an implementation will build a compact representation of an *a.i.* 's execution trace or even bypass the trace and construct a representation of the collecting semantics directly—the cache computed by a flow analysis is a classical example [3]; the "cache" computed by denotational-semantics analysis is another [28].

We begin by developing these notions for the operational semantics of flowchart languages.

2 Abstract Interpretation of Flowchart Programs

The principles of abstract interpretation were established for flowchart programs by Cousot and Cousot [11], and most of the material in this section is a review of their work. Precedents for the use of traces as seen in this section are found in [16, 32, 31].

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, $v \vdash pp$, and state transitions are listed in

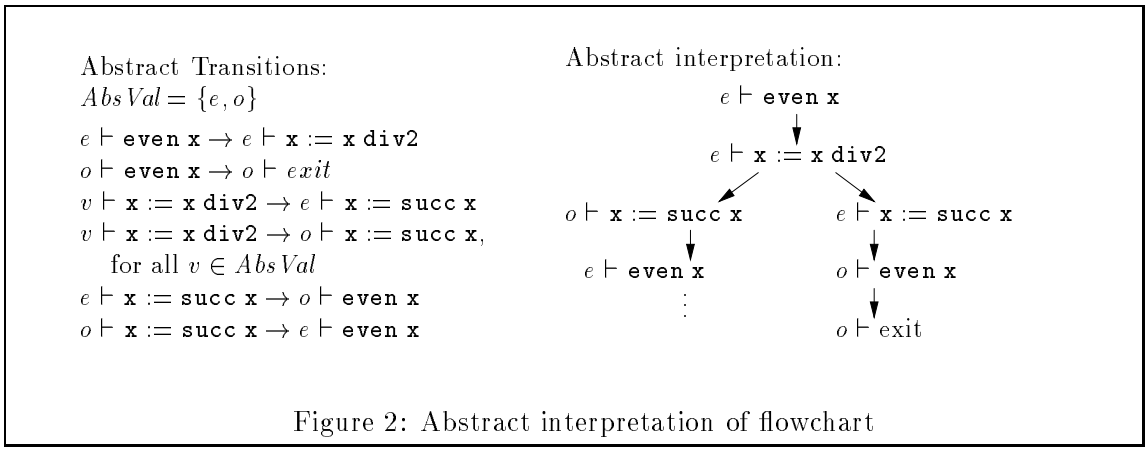


Figure 2: Abstract interpretation of flowchart

the middle column of the Figure. The program's *c.i.* is drawn as a trace; since the program is deterministic, the trace 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 *a.i.* 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 is revised into one or more abstract transitions. The resulting abstract semantics must be nondeterministic in its interpretation of `div2`. This implies that the *a.i.* should be a set of traces, but we represent the set by a single, nondeterministic, trace tree. Thus, the program's *a.i.* contains more paths than what appear in the *c.i.* Also, the *a.i.* trace is infinite, but the infinite path contain a repetition node, meaning that the tree is *regular* and has the finite representation shown in the Figure—termination of the *a.i.* is not a problem here, because 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 transition relation in Figure 1 to the abstract one in Figure 2. Let $\beta : Val \rightarrow AbsVal$ 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, and $c \in Val$,

$$c \vdash pp \rightarrow c' \vdash pp' \text{ implies there exists } a' \in AbsVal$$
$$\text{such that } \beta(c) \vdash pp \rightarrow a' \vdash pp' \text{ and } \beta(c') \sqsubseteq a'$$

The inequality, $\beta(c') \sqsubseteq a'$, is a weakening of the expected $\beta(c') = a'$ because an acceptable *a.i.* can lose precision. For example, we might code the `div2` operation in Figure 2 so that it is deterministic: $a \vdash$ x := xdiv2 $\rightarrow \top \vdash$ x := succ x, for all $a \in AbsVal$, where $\top$ represents "either even or odd." The extra element necessitates an *approximation ordering* [13] on $AbsVal = \{e, o, \top\}$: $a \sqsubseteq \top$ and $a \sqsubseteq a$, for all $a \in AbsVal$. Then, we require that the transition relation is *monotonic* with respect to the ordering:

$$a_1 \vdash pp \rightarrow a_1' \vdash pp' \text{ and } a_1 \sqsubseteq a_2 \text{ imply } a_2 \vdash pp \rightarrow a_2' \vdash pp' \text{ and } a_1' \sqsubseteq a_2'$$

Momentarily, we will see that existence of the homomorphism property ensures that a program's *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

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

that is, c is safely approximated (or represented) by a. 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.

Since a trace is a tree of transitions, we will write $root(t)$ to denote the start state of trace t. If there is a transition, $v \vdash pp \rightarrow v' \vdash pp'$, and $root(t) = v' \vdash pp'$, we write $c \vdash pp \longrightarrow t$ to denote the composite trace. Finally, because of the nondeterminism in trace trees, we generalize the above notation to sets of transitions and traces: if $\{v \vdash pp \rightarrow v_i \vdash pp_i\}_{1 \leq i \leq n}$ is a set of transitions from the state $v \vdash pp$, and $\{t_i \mid root(t_i) = v_i \vdash pp_i\}_{1 \leq i \leq n}$ is a set of traces, we write $c \vdash pp \longrightarrow \{t_i \mid root(t_i) = v_i \vdash pp_i\}_{1 \leq i \leq n}$ to denote the composite nondeterministic trace tree.

A program's *c.i.*, t_C, is *safely approximated* (or *simulated*) by an *a.i.*, t_A, iff $t_C \; safe_{Trace} \; t_A$, where

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

The intent of $safe_{Trace}$ is that every computation path in t_C is safely approximated by one in t_A. The consequences of this property will be studied later.

A technical issue is that the definition of $safe_{Trace}$ 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 program p and input $c \in Val$, let $trace_C(p_0, c)$ be p's *c.i.*, where p_0 is p's entry program point; similarly, $trace_A(p_0, a)$ is the program's *a.i.*, for $a \in AbsVal$.[1] Then, $c \; safe_{Val} \; a$ implies $trace_C(p_0, c) \; safe_{Trace} \; trace_A(p_0, a)$, when the following *relational homomorphism property* holds for the concrete and abstract transition relations:

$$c \; safe_{Val} \; a \text{ and } c \vdash pp \rightarrow c' \vdash pp' \text{ imply there exists } a' \in AbsVal$$
$$\text{such that } a \vdash pp \rightarrow a' \vdash pp' \text{ and } c' \; safe_{Val} \; a'$$

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

From here on, we work entirely with the relational representations; alternative frameworks are discussed at length in [13]. 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 $c \; safe_{Val} \; a'$ [13, 43, 58].

[1] The definitions for $trace_C(p_0, c)$ and $trace_A(p_0, a)$ are in Section 2.3.

2.2 Inductively and Coinductively Defined Sets

The flowchart traces in the previous section can be infinite, and proofs on infinite traces are best worked with coinductive techniques [2, 54, 40], which we now review. The following presentation is summarized from Cousot and Cousot [14].

We begin with the classical inductive definition. Let $\mathcal{U}$ be a universe of terms, and let $F{:}\mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$ be continuous[2] with respect to the powerset lattice $\langle \mathcal{P}(\mathcal{U}), \subseteq \rangle$. The *set defined inductively by* F is $lfpF = \bigcup_{i \geq 0} S_i$, where $S_0 = \{\}$ and $S_{i+1} = F(S_i)$. Note also that $lfpF = \bigcap\{S' \mid closed_F S'\}$, where $closed_F S'$ iff $F(S') \subseteq S'$. That is, $lfpF$ is the smallest closed set. The latter definition gives a standard reasoning technique, *fixed point induction*: to prove $lfpF \subseteq P$, that is, every element of $lfpF$ has property P, it suffices to find a set $S' \subseteq P$ such that $closed_F S'$. When F is defined from a BNF rule, then proving $closed_F P$ 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 $gfpF = \bigcap_{i \geq 0} T_i$, where $T_0 = \mathcal{U}$ and $T_{i+1} = F(T_i)$. Also, $gfpF = \bigcup\{T' \mid dense_F T'\}$, where $dense_F T'$ iff $T' \subseteq F(T')$. That is, $gfpF$ is the largest dense set. This gives the reasoning technique of *fixed point coinduction*: to prove $Q \subseteq gfpF$, it suffices to find a set, Q', such that $Q \subseteq Q'$ and $dense_F Q'$. When a property, P, is defined coinductively as $P = gfpF$, then proving $dense_F(gfpG)$ is a standard way of proving that coinductively defined set $gfpG$ 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 $\bar{V} : \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$; $\bar{V}(S) = \{0\} \cup \{1s \mid s \in S\}$; we obtain $lfp\bar{V} = \{1^n 0 \mid n \geq 0\}$, whereas $gfp\bar{V} = lfp\bar{V} \cup \{1^\omega\}$.

It is helpful to think of strings as traces with a single path; when calculating $lfp\bar{V}$, S_i contains those traces of length i or less that are certified members of $lfp\bar{V}$; in contrast, T_i contains those traces that are certified as far as length i and are not yet excluded from membership in $gfp\bar{V}$.

Say that we wish to prove that all strings in $lfp\bar{V}$ are finite: by fixed-point induction, we need only show that the set $is_finite \subseteq \mathcal{U}$ is closed: $\bar{V}(is_finite) \subseteq is_finite$. This is the usual structural induction proof. A fixed-point coinduction typically involves recursively defined predicates: say that we wish to show, for all strings (trees) in $gfp\bar{V}$, that no 1 follows a 0. Define these predicates:

$$ok(s) \text{ iff } zeroes(s) \text{ or } s = 1 \text{ or } (s = 1t \text{ and } ok(t))$$
$$\text{where } zeroes(s) \text{ iff } s = 0 \text{ or } (s = 0t \text{ and } zeroes(t))$$

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

For the remainder of this paper, we use a universe, $\mathcal{U}$, of finitely branching trees of at most countably infinite (ω-) depth[23, 25].

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

2.3 Coinduction Applied to Concrete and Abstract Interpretations

An execution trace is an element of a (co)inductively defined set, which we now define. Here is the specification of a well formed trace (*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 traces are the finite ones; a coinductive interpretation includes the countably infinite traces. We use the coinductive interpretation.

For program p with entry point p_0 and input v_0, it is traditional to generate its trace, $trace(p_0, v_0)$, by working from the start state, $v_0 \vdash p_0$, and expanding all possible transitions. Some auxiliary notation is needed to make this precise: If t is an incomplete trace, l is a leaf in t, and t' is a trace such that $root(t') = l$, then we write $[t'/l]t$ to denote the replacement of l by trace t'. A set of such substitutions is written $[t'_i/l_i]_{i \in I}t$.

The generation of $trace(p_0, v_0)$ is formalized in stages, $t_i, i \geq 0$:

- $t_0 = v_0 \vdash p_0$
- $t_{k+1} =$ for each leaf, $l_i = (v_i \vdash pp_i), i \in I$, in t_k,
  let $\{v_i \vdash pp_i \rightarrow v_{ij} \vdash pp_{ij}\}_{1 \leq ij \leq i_n}$ be all possible transitions from l_i,
  in $[v_i \vdash pp_i \longrightarrow \{v_{ij} \vdash pp_{ij}\}_{1 \leq ij \leq i_n}/l_i]_{i \in I}t_k$

Clause 2 states that all leaves in t_k are expanded by all possible one-step transitions to generate t_{k+1}. Finally, define $trace(p_0, v_0) = lim_{i \geq 0}\, t_i$, which is a well-defined trace.[3]

Both the *c.i.* and the *a.i.* of a program are defined in the above fashion. Next, the safety relation, $safe_{Trace}$, is defined coinductively, and we can now prove the simulation property: for inputs $c \in Val$, $a \in AbsVal$, $c\ safe_{Val}\ a$ implies $trace(p_0, v)\ safe_{Trace}\ trace(p_0, a)$.

The proof proceeds as follows: First, note that $safe_{Trace} = gfpF(S) = \{(t, t') \mid root(t)\ safe_{State}\ root(t')$ and for all $root(t) \longrightarrow t_i$, there exists $root(t') \longrightarrow t'_j$ such that $S(t_i, t'_j)\}$. Let wft_C and wft_A denote the set of well-formed concrete and abstract traces respectively, and 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 $(trace(p_0, c_0), trace(p_0, a_0)) \in S_0$, 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 $root(t) \longrightarrow t_i$, 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 trace $t'_j \in wft_A$, implying that $root(t') \longrightarrow t'_j$. Finally, it is immediate that $(t_i, t'_j) \in S_0$.

[3]This is proved by fixed point coinduction: we note that $wft = gfpW$, where $W(S) = \{t \mid$ (i) $t = (v \vdash pp)$, or (ii) $t = (v \vdash pp \longrightarrow \{t_i\}_{i \in I})$ and $\{v \vdash pp \rightarrow v_i \vdash pp_i\}_{i \in I}$ are all possible transitions from $v \vdash pp$ and for all $i \in I, t_i \in S$ and $root(t_i) = (v_i \vdash pp_i)\}$. Consider the set $S_0 = \{t \mid t$ is a subtree of $trace(p_0, v_0)\}$; the result follows from that proof that $S_0 \subseteq W(S_0)$. The key to the proof is that every $t \in S_0$ has $root(t) = v \vdash pp$ that was created as a leaf at some stage, t_k, implying that at stage t_{k+1}, $v \vdash pp \longrightarrow \{v_i \vdash pp_i\}_{i \in I}$, where each $v_i \vdash pp_i$ is itself the root of a trace in S_0.

2.4 A Comparison with Mathematical Induction

It is useful to consider how the above proof resembles a proof done by induction on the length of the trace. For simplicity, consider deterministic traces (sequences) only and an arbitrary safety relation, $\mathcal{R}$. The claim that concrete trace $t_C = C_0 \to C_1 \to \cdots \to C_i \to \cdots$ is simulated by abstract trace $t_A = A_0 \to A_1 \to \cdots \to A_i \to \cdots$ is defined as $\forall i \geq 0, C_i \; \mathcal{R} \; A_i$; the induction proof 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 by coinduction, these two steps will reappear, but some startup machinery is required: The universally quantified safety property is recoded recursively as $safe = gfpF$, where $F(S) = \{(t, t') \mid head(t) \; \mathcal{R} \; head(t') \text{ and } (tail(t), tail(t')) \in S\}$. 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 example was meant to emphasize the similarities between mathematical induction and coinduction proof techniques, one notes also 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 trace structures and their properties grow in complexity, it becomes more convenient to work with coinduction—safety properties stay simple and proofs stay short.

2.5 How to Derive the Abstract Semantics from the Concrete One

Once the abstract domain, $AbsVal$, is selected, we wish to derive the abstract semantics from the concrete one so that the relational homomorphism property holds. For each program point, pp, we define the abstract transition rule

$$a \vdash pp \to a' \vdash pp' \text{ if there exists } c \in Val \text{ such that } c \; safe_{Val} \; a, c \vdash pp \to c' \vdash pp', \text{ and } c' \; safe_{Val} \; a'$$

The above condition is sufficient, but not necessary, for a relational homomorphism: If $AbsVal$ is partially ordered, $safe_{Val}$ is U-closed, and $c' \; safe_{Val} \; \sqcap A$, where $A = \{a' \mid c' \; safe_{Val} \; a'\}$, then one obtains a better quality analysis by using $a \vdash pp \to \sqcap A \vdash pp'$.[4]

2.6 Liveness Abstract Interpretations

The examples so far are oriented towards safety analyses, where an *a.i.* contains more transitions in its trace than does the corresponding *c.i.* A *liveness analysis* is the dual: An *a.i.* contains a transition only if all corresponding *c.i.*s possess a corresponding transition. Liveness analyses are of primary interest when one wishes to validate properties such as starvation freedom.

[4]Indeed, these conditions suffice for defining a *Galois connection* between $\mathcal{P}(Val)$ and $AbsVal$, for which there is extensive advice for deriving precise analyses [12, 13, 39, 44, 58].

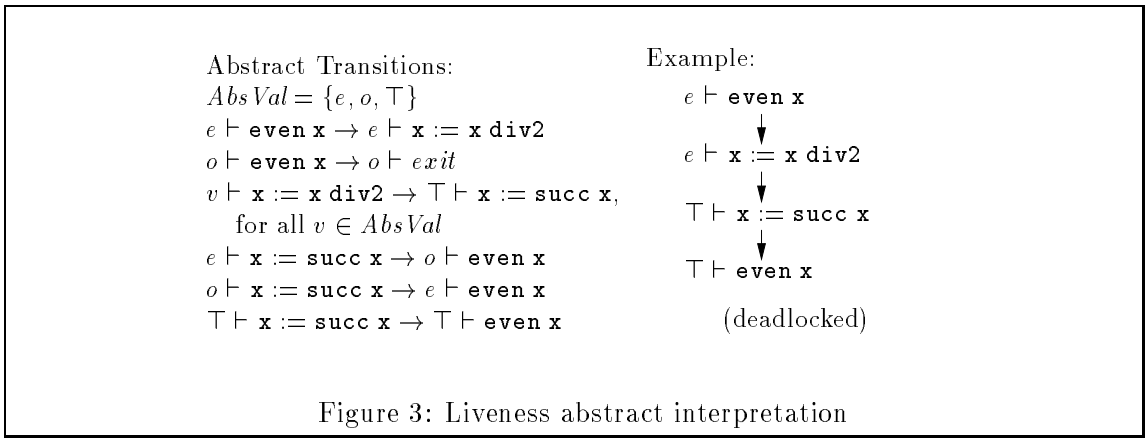


Figure 3: Liveness abstract interpretation

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 traces is expressed as follows:

$$t\ live_{Trace}\ t' \text{ iff } root(t)\ live_{State}\ root(t'), \text{ and}$$
$$\text{for every transition, } root(t') \longrightarrow t'_j,$$
$$\text{there exists a transition, } root(t) \longrightarrow t_i, \text{ such that } t_i\ safe_{Trace}\ t'_j$$

That is, the *c.i.* is a simulation of the *a.i.* .

Figure 3 shows the concrete semantics of Figure 1 naively abstracted for a liveness analysis. Unfortunately, the reuse of $AbsVal$ from Figure 2 produces an uninteresting liveness analysis that can analyze only one loop iteration—the problem is the abstract transition rule for `x:=x div2`, which cannot give a precise output. At best, a $\top$-value can be used, and this leads to deadlock at the loop's test. Selecting the appropriate abstract domains for liveness analysis is a little-understood art.

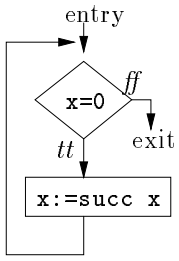
To prove the liveness relation between the *c.i.* and the *a.i.*, the (dual of the) relational homomorphism property is required, and this can be obtained by deriving the abstract semantics from the concrete one as follows:

$$a \vdash pp \rightarrow a' \vdash pp' \text{ only if for all } c \in Val,$$
$$c\ live_{Val}\ a \text{ implies } c \vdash pp \rightarrow c' \vdash pp', \text{ and } c'\ live_{Val}\ a'$$

2.7 Termination of the Abstract Trace

The *a.i.* in Figure 2 is infinite, but its construction is finite because a state repeats in the infinite path; the trace is a *regular* tree and can be represented by a finite one with backwards arc(s). Unfortunately, there is no guarantee that every *a.i.* is a regular tree: for example, 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" or "cache" of program points and the inputs that arrive at those points. This concept is realized within *a.i.* as a *memoization* [58, 59] or *widening* [11] of the abstract trace.

Figure 4 shows a constant propagation analysis with memoization. When a program point repeats in the trace, all previous inputs to the point are joined with the newest one, and the trace proceeds. This forces termination but with loss of precision. The memoized



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

Memoized a.i.:

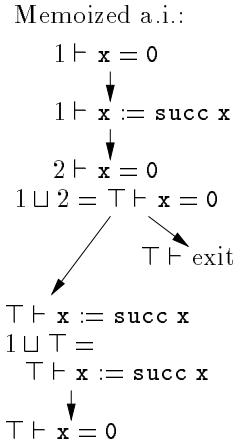


Figure 4: Memoized abstract interpretation

trace can be defined in stages, like before:

- $t_0 = v_0 \vdash p_0$
- $t_{k+1} =$ for each leaf, $l_i = (v_i \vdash pp_i), i \in I,$ in t_k,
  let $\{v_i \vdash pp_i \rightarrow v_{ij} \vdash pp_{ij}\}_{1 \leq ij \leq i_n}$ be all possible transitions from l_i;
  and for each pp_{ij}, let $V_{ij} = \sqcup\{v' \mid v' \vdash pp_{ij}$ appears in $t_k\}$
  in $[v_i \vdash pp_i \longrightarrow \{v_{ij} \sqcup V_{ij} \vdash pp_{ij}\}_{1 \leq ij \leq i_n} / l_i]_{i \in I} t_k$

Clause 2 states how all previous inputs to a program point, p_{ij}, are joined with the newest input, v_{ij}, to make a new leaf, $v_{ij} \sqcup V_{ij} \vdash pp_{ij}$, in the trace.

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 abstract operations are monotone on $AbsVal$. Monotonicity ensures that for each program point, pp, the sequence of states, $a_i \vdash pp$, $i \geq 0$, occurring along a path in the *a.i.* 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 $safe_{Trace}$ is U-closed. (The proof goes by coinduction.)

2.8 Collecting Semantics: First-Order and Second-Order

Once a program's trace is constructed, whether it is a *c.i.* or an *a.i.*, information must be extracted from it for validation or code improvement. The extracted information is called the *collecting semantics*.

The classic collecting semantics is *first order*: It associates to each program point the set of input values that appeared at the program point in the trace [11, 45]: for trace, t, $coll_t : ProgramPoint \rightarrow \mathcal{P}(Val)$ is 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 *c.i.* , but it is equally applicable to an *a.i.* , and we see in Section 2.8 that an iterative data-flow analysis calculates exactly the collecting semantics of a memoized *a.i.* An *a.i.* 's collecting semantics is sometimes weakened by joining the abstract values for a program point: $coll'_t(pp) = \sqcup coll_t(pp)$, since in practice this is easier to calculate and often suffices for code improvement applications.

If the usual safety result has been proved, that is, the abstract semantics simulates the concrete semantics, then it follows that the collecting semantics for the *a.i.* safely approximates the collecting semantics of the corresponding *c.i.* , which is the "fundamental theorem" of abstract interpretation: for program, p, if c $safe_{Val}$ a, then for all $pp \in ProgramPoint$,

$$coll_{trace(p_0,c)}(pp) \subseteq \gamma(coll_{trace(p_0,a)}(pp)),$$

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 dual result holds for liveness analysis.

Perhaps more important but less well understood is *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 collecting semantics as follows:

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

Notable applications of second-order collecting semantics are available-expression and live-variable data-flow analyses, which are respectively forwards and backwards, but second-order collecting semantics lie at the foundations of model-checking, as well; this application is examined below.

Finally, Cousot and Cousot [13] suggest that the collecting semantics of a trace can be any property or set of properties expressed in a logic, $\mathcal{L}$. Given a trace, 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 a concrete and abstract interpretations, and we assume that the same $\mathcal{L}$ can be used for both concrete and abstract traces.

With this approach, we must first prove a weak consistency relation between the safety relation, $safe_{Trace}$, and $\mathcal{L}$:

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

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

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

To have weak completeness, there must be a close—or even exact—match between $\mathcal{L}$ and $AbsVal$.

The two above notions are titled "weak" because decidability is lacking: t_C $safe_{Trace}$ t_A and $t_A \not\models \phi$ does *not* imply $t_C \models \neg\phi$. 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, but the price one pays is either an $AbsVal$ set that differs little from Val or a low-precision definition of $\mathcal{L}$.

These notions of soundness and completeness are developed by Dams in his thesis [17].

2.9 Representations of the Collecting Semantics

If the purpose for calculating an *a.i.* is to obtain an abstract collecting semantics for program points, then an implementation can generate the *a.i.* implicitly while calculating explicitly a representation of the collecting semantics. Typically, 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 [28, 10]; and the solution of a constraints set generated for type inference [4, 5, 67] or control-flow analysis [27, 52].[5]

Because of the emphasis placed upon the collecting semantics, it is all too easy to confuse an *a.i.* with the collecting semantics extracted from it. As a result, precision can be inadvertantly lost when an algorithm for calculating directly the collecting semantics is formulated before the *a.i.* upon which it is based. Also, safety proofs are complicated when they are worked on the collecting semantics algorithm rather than upon the *a.i.* .

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.

2.10 Application: Data-Flow Analysis

A standard iterative data-flow analysis encodes a program and its data flow as a set of simultaneous equations, one equation per program point. The equations are solved with a least fixed-point iteration [3]. As noted in the previous section, a data-flow analysis calculates a representation of a collecting semantics.

For example, the collecting semantics of the even-odd analysis of the program in Figure 1 is encoded with flow equations named in_{pp}, for each $pp \in ProgramPoint$, of the form

$$in_{pp} = \bigsqcup_{q \in pred\ pp} f_q(in_q)$$

where $AbsVal = \{\bot, e, o, \top\}$, $f_{\texttt{even x}}(v) = v$, $f_{\texttt{x:=x div2}}(v) = \top$, and $f_{\texttt{x:=succ x}}(e) = o$, $f_{\texttt{x:=succ x}}(o) = e$, and $f_{\texttt{x:=succ x}}(\top) = \top$.

An equation, in_{pp}, defines the data flow into pp; to initialize, an extra equation is written for the program's entry point: $in_{entry} = e$.

Figure 5 shows the solution of the equations for the example. The process starts from $\bot$-elements, and a *computational partial ordering* [13], which in this case coincides with the approximation ordering, is used to calculate the join operation, $\sqcup$, and solve the equations. It takes little work to prove that the solution of the data-flow equations is exactly the first-order collecting semantics of the program's memoized *a.i.* : Column i of the table in the Figure 5 equals the collecting semantics of stage i of the memoized *a.i.* in Figure 4.[6]

Many flow analyses—available expressions and live variables, for example—are second-order, because the analyses must calculate execution paths containing histories of expression

[5]Contrast this with the classic formulation of strictness analysis [6], which is a true *a.i.* and *not* a calculation of a collecting semantics.

[6]Indeed, the collecting semantics of an *a.i.* is known historically as the *meet-over-all-paths* analysis (MOP), whereas the collecting semantics of a memoized *a.i.* is known as the *maximal fixed-point* analysis (MFP) [45].

$in_{entry} = e$
$in_{\texttt{even x}} = in_{entry} \sqcup f_{\texttt{x:=succ x}}(in_{entry})$
$in_{\texttt{x:=x div2}} = in_{\texttt{even x}}$
$in_{\texttt{x:=succ x}} = f_{\texttt{x:=x div2}}(in_{\texttt{x:=x div2}})$
$in_{exit} = in_{\texttt{even x}}$

iteration	0	1	2	3	4	5	6	7
entry	$\bot$	e	e	e	e	e	e	e
`even x`	$\bot$	$\bot$	e	e	e	$\top$	$\top$	$\top$
`x:=x div2`	$\bot$	$\bot$	$\bot$	e	e	e	$\top$	$\top$
`x:=succ x`	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$
exit	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$

Figure 5: Iterative data-flow analysis

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

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

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

$$[\![p]\!]\rho = \{s \mid s \models p\}$$
$$[\![\neg p]\!]\rho = \{s \mid s \not\models p\}$$
$$[\![\phi_1 \wedge \phi_2]\!]\rho = [\![\phi_1]\!]\rho \cap [\![\phi_2]\!]$$
$$[\![\phi_1 \vee \phi_2]\!]\rho = [\![\phi_1]\!]\rho \cup [\![\phi_2]\!]$$
$$[\![\Box\phi]\!]\rho = \{s \mid \text{for all } s' \text{ such that } s \to s', s' \in [\![\phi]\!]\rho\}$$
$$[\![\Diamond\phi]\!]\rho = \{s \mid \text{there exists } s' \text{ such that } s \to s' \text{ and } s' \in [\![\phi]\!]\rho\}$$
$$[\![\mu Z.\phi]\!]\rho = \bigcup_{i \geq 0} S_i, \text{ where } \begin{cases} S_0 = \emptyset \\ S_{i+1} = [\![\phi]\!]([Z \mapsto S_i]\rho) \end{cases}$$
$$[\![\nu Z.\phi]\!]\rho = \bigcap_{i \geq 0} S_i, \text{ where } \begin{cases} S_0 = \mathit{State} \\ S_{i+1} = [\![\phi]\!]([Z \mapsto S_i]\rho) \end{cases}$$
$$[\![Z]\!]\rho = \rho(Z)$$

Figure 6: Mu-calculus syntax and semantics

evaluation and futures of variable use. These flow analyses must calculate representations of the paths, namely, sets of available expressions and sets of live variables. In this fashion, a representation of a second-order collecting semantics is calculated. Second-order data-flow analyses are intimately related to model checking, which we now examine.

2.11 Application: Model Checking

Model checking is a technique for validating properties of paths in a program's trace [7, 17, 38]. The technique is used primarily to validate safety and liveness properties of circuits and protocols, but it is applicable to validating finite-state traces of programs, which can be obtained by *a.i.* [8, 17].

Properties are stated in a logic, $\mathcal{L}$, of which CTL* [7] and mu-calculus [65] are commonly used; we employ the latter. Figure 6 defines the syntax and semantics of the 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; 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 trace in Figure 2 can be model checked for simple path properties—for example, one can verify that all paths from the trace's root must include the command `x:=succ x` by checking the proposition $\mu Z.(pp = \texttt{x} := \texttt{succ x}) \vee \Box Z$, 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 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 example program with input 2.)

It is easy to prove that model checking upon a safe *a.i.* is (weakly) consistent when the $\Diamond$ operator is removed from the calculus; call the result the *box-mu-calculus*. Dually, the *diamond-mu-calculus* can be used to model check a proved-live *a.i.*

Here, the collecting semantics of a safe *a.i.* are those propositions in the box-mu-calculus that hold for the root of the trace. The collecting semantics is fundamentally second-order.

Finally, it is striking that second-order data-flow analyses can be encoded as propositions in the mu-calculus [19, 63, 64]; 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 exactly 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 [37] 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 the entry to program point, pp. The flow equations are solved with a greatest fixed point calculation: the initial approximation are 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 state:

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

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.[7]

Rather than working with the control flow graph, one can obtain higher precision model check by working with a less trivial *a.i.* of the flowchart—the model checker calculates a second-order collecting semantics of the *a.i.* . Clarke, Grumberg, and Long use this technique for circuit and protocol validation [8].

[7]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.

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

$$e \ ::= \ op(e_i)_{i \in 1..n} \ \mid \ \texttt{if}\ e_1\ e_2\ e_3 \ \mid \ \texttt{rec}\ f\ x.\ e \ \mid \ e_1\ e_2 \ \mid \ x$$

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

$$\frac{\{\rho \vdash e_i \Downarrow v_i\}_{i \in 1..n}}{\rho \vdash op(e_i)_{i \in 1..n} \Downarrow f_{opC}(v_i)_{i \in 1..n}} \qquad \frac{\rho \vdash e_1 \Downarrow tt \quad \rho \vdash e_2 \Downarrow v}{\rho \vdash \texttt{if}\ e_1\ e_2\ e_3 \Downarrow v} \qquad \frac{\rho \vdash e_1 \Downarrow ff \quad \rho \vdash e_3 \Downarrow v}{\rho \vdash \texttt{if}\ e_1\ e_2\ e_3 \Downarrow v}$$

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

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

Figure 7: Concrete big-step semantics

There is a correspondence in the other direction as well: The standard algorithm for checking CTL (or mu-calculus without alternating fixed-point quantifiers) translates a CTL proposition into a first-order flow equation set and solves iteratively [21].

3 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 [36, 51], where a language's semantics is the set of derivations generated inductively from a set of inference rule schemes. Figure 7 gives the concrete semantics of an untyped, higher-order functional language where all user-defined abstractions are recursive.[8] Primitive operations, *op*, are interpreted as functions, f_{opC}, on *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-pointing arrow. Figure 8 shows a *c.i.* of a convergent program that uses two primitive operations, `even` and `div2`, whose interpretations are given in the Figure.

Figure 9 gives the abstract semantics for an even-odd analysis for the language in Figure 7. The abstract semantics must reinterpret the primitive functions, f_{opA}, on *AbsVal*, and ideally the inference rules are modified in no other way. But problems arise with nondeterminism: For example, if the language possessed the rules $\frac{\rho \vdash e_1 \Downarrow v_1}{\rho \vdash e_1\ \texttt{or}\ e_2 \Downarrow v_1}$ and

[8] The problems addressed in this section are not unique to functional languages; a while-loop language with procedures behaves similarly [51].

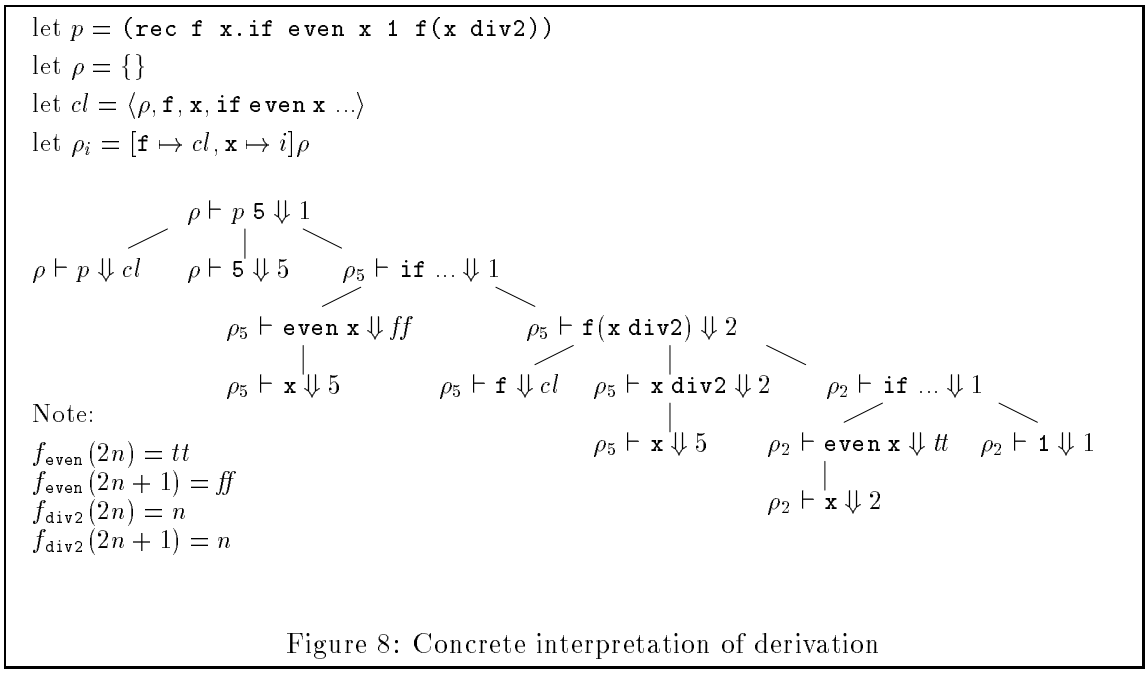


Figure 8: Concrete interpretation of derivation

$\frac{\rho \vdash e_2 \Downarrow v_2}{\rho \vdash e_1\ \mathtt{or}\ e_2 \Downarrow v_2}$, then a *c.i.* for e_1 `or` e_2 would use just one of the rules, but a safe *a.i.* must employ both. This suggests that the *a.i.* should be a set of derivations, but it is traditional to encode the set into a single, nondeterministic, derivation tree. Working with a single tree forces us to join the synthesized attributes, v_1 and v_2, in effect generating a new rule scheme for the *a.i.* : $\frac{\rho \vdash e_1 \Downarrow v_1 \quad \rho \vdash e_1 \Downarrow v_1}{\rho \vdash e_1\ \mathtt{or}\ e_2 \Downarrow v_1 \sqcup v_2}$. This issue arises again with `if`: when its test, e_1, cannot be resolved to *tt* or *ff* (we momentarily use $\top$ to denote this situation), then both e_2 and e_3 must be interpreted and their values joined.[9]

Figure 10 displays the *a.i.* 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—we must interpret the abstract semantics coinductively. Also, the synthesized attribute, a, for the repeated state, $\rho_\top \vdash \mathtt{if} \ldots \Downarrow a$, is unresolved. The equality $a = o \sqcup a$ must be satisfied, which suggests that 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. We tackle these issues in turn.

3.1 Safety Properties of Finite and Infinite Derivations

For the moment, we backtrack and assume that both concrete and abstract semantics are defined inductively. Thus, for a universe, $\mathcal{U}$, of finitely-branching trees, the set of well-formed derivation trees derived from a set of inference rules, $\mathcal{R}$, is the least set satisfying

[9]This raises the issue of the approximation ordering on $AbsVal$. The definitions of the four sets in Figure 9 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 $a \in AbsVal$; the ordering on $AbsEnv$ is pointwise; and $AbsClos$'s ordering is defined componentwise (Id and $Expr$ are ordered discretely).

$$
\begin{array}{l}
v \in \mathit{AbsVal} = (\mathit{AbsNat} \cup \mathit{Bool} \cup \mathit{AbsClos})^{\top} \\
\qquad \text{such that } v \sqsubseteq \top, \text{ for all } v \in \mathit{AbsVal} \\
n \in \mathit{AbsNat} = \{e, o\} \\
\langle \rho, x, f, e \rangle \in \mathit{Clos} = \mathit{AbsEnv} \times \mathit{Id} \times \mathit{Id} \times \mathit{Expression} \\
\rho \in \mathit{AbsEnv} = \mathit{Id} \stackrel{fin}{\rightarrow} \mathit{AbsVal}
\end{array}
$$

Semantics rules for `if`, `rec`, $(e_1\ e_2)$, and x carry over from Figure 7. Replace the rule for `op` and add one rule for `if` as follows:

$$
\frac{\{\rho \vdash e_i \Downarrow v_i\}_{i \in 1..n}}{\rho \vdash op(e_i)_{i \in 1..n} \Downarrow f_{opA}(v_i)_{i \in 1..n}} \qquad \frac{\rho \vdash e_1 \Downarrow \top \quad \rho \vdash e_2 \Downarrow v_2 \quad \rho \vdash e_3 \Downarrow v_3}{\rho \vdash \texttt{if}\ e_1\ e_2\ e_3 \Downarrow v_2 \sqcup v_3}
$$

Figure 9: Abstract big-step semantics

the predicate $\mathit{wftree}_{\mathcal{R}} \subseteq \mathcal{U}$:

$$
\begin{array}{l}
\mathit{wftree}_{\mathcal{R}}(t) \text{ iff there exists } \frac{s_1, \ldots, s_n}{\mathit{root}(t)} \in \mathcal{R}, n \geq 0, \\
\text{and for all child subtrees, } t_i, i \in 1 \ldots n, \text{ of } t, \mathit{root}(t_i) = s_i \text{ and } \mathit{wftree}_{\mathcal{R}}(t_i)
\end{array}
$$

For simplicity, $\mathcal{R}$ is a set of rules, rather than rule schemes.

As before, a safety relation must be 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

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

Note that safe_{Val} is U-closed, which is required. Of course, the relational homomorphism property must hold for corresponding operations f_C and f_A: if $c_i\ \mathit{safe}_{Val}\ a_i$, for all $i \in 1..n$, then $f_C(c_i)_{i \in 1..n}\ \mathit{safe}_{Val}\ f_A(a_i)_{i \in 1..n}$.

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_A, there exists a child subtree t_j of t_A such that $t_i\ \mathit{safe}_{Tree}\ t_j$.[10] The intuition is that every computation path in t_C is safely approximated by some path 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 $t_C \in \mathit{wftree}_{\mathcal{C}}$

[10] Note that j need not equal i, e.g., consider the *c.i.* and *a.i.* for `if e1 e2 e3`.

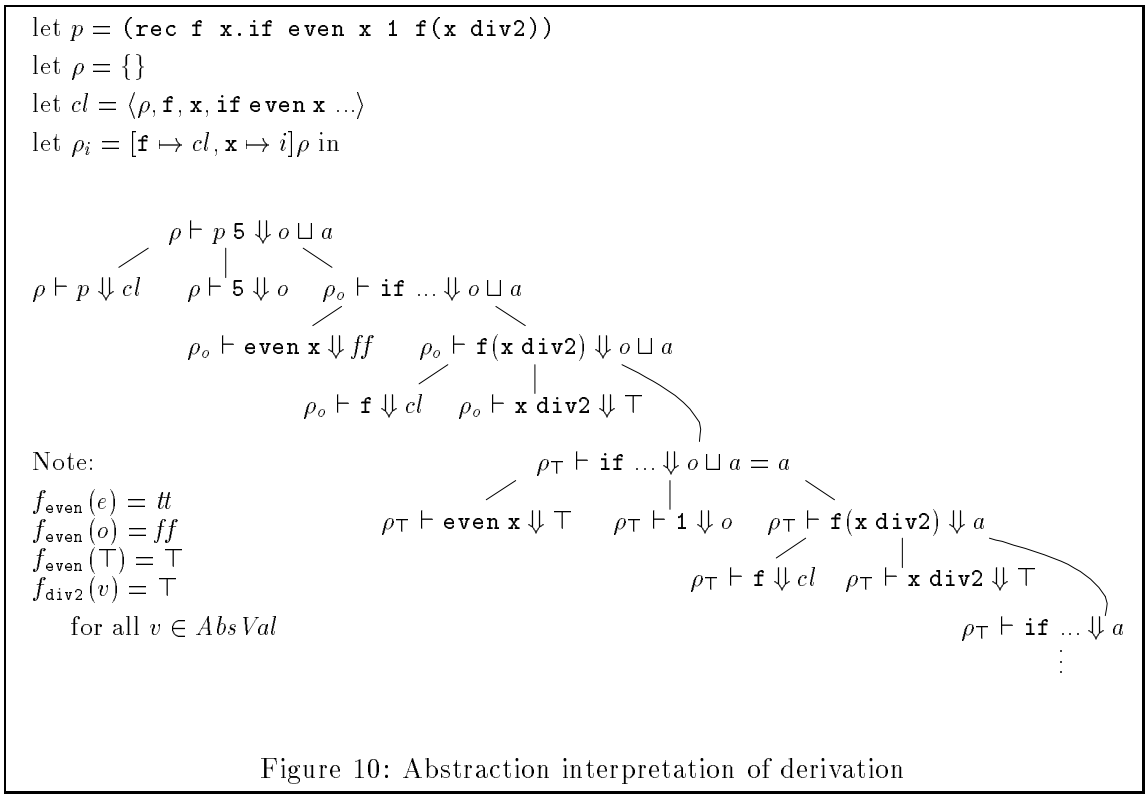


Figure 10: Abstraction interpretation of derivation

such that $root(t_C) = \rho_C \vdash p \Downarrow c$, for every $t_A \in wftree_{\mathcal{A}}$ such that $root(t_A) = \rho_A \vdash p \Downarrow a$, it is the case that $t_C\ safe_{Tree}\ t_A$. The proof comes easily by induction on the height of the concrete derivation tree; see [26, 51] for example proofs in this style.

3.2 Infinite Abstract Derivations

We desire that every program with a *c.i.* also possess an *a.i.* , and Figure 10 makes clear that infinite abstract derivations are necessary. This implies that the abstract semantics rule set, $\mathcal{A}$, defines by coinduction $wftree_{\mathcal{A}}$, which includes infinite well-formed derivations. Unfortunately, because of the synthesized attributes in the sequents, the coinductively defined set also includes multiple derivations for a program, p, and its initial ρ_A—an example appears in Figure 10, where fixing $a = o$ 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.[11]

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 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.

3.3 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

[11] This requires that the inference rules are monotone with respect to the ordering.

$$\frac{\{\rho \vdash e_j \Downarrow v_j\}_{j \in 1..k-1} \quad \rho \vdash e_k \Uparrow, \text{ for some } k \leq n}{\rho \vdash op(e_i)_{i \in 1..n} \Uparrow} \qquad \frac{\rho \vdash e_1 \Uparrow}{\rho \vdash \texttt{if}\ e_1\ e_2\ e_3 \Uparrow}$$

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

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

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

Figure 11: Negative big-step inference rules

that a divergent program, p, and initial environment, ρ_C, might have multiple, well-formed, infinite derivations; a simple example comes from the rule, $\frac{\rho \vdash \texttt{loop} \Downarrow v}{\rho \vdash \texttt{loop} \Downarrow v}$, 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 [58]. 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 [14] 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 7; Figure 11 shows the negative rules. For example, the first rule in Figure 11 states, if convergent derivations exist for arguments, e_i, $i \in 1..k-1$, but e_k is divergent, then so is $op(e_i)_{i \in 1..n}$.

The positive rules define the set of positive trees:

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

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

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

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

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

- $(\rho_C \vdash e \Downarrow c)$ $safe_{Seq}$ $(\rho_A \vdash e \Downarrow a)$ iff ρ_C $safe_{Env}$ ρ_A and c $safe_{Value}$ a
- $(\rho_C \vdash e \Uparrow)$ $safe_{Seq}$ $(\rho_A \vdash e \Downarrow a)$ iff ρ_C $safe_{Env}$ ρ_A

The definition of $safe_{Tree}$ remains as before: $safe_{Tree} = gfp\ F$, where $F(S) = \{(t_C, t_A) \mid root(t_C)\ safe_{Seq}\ root(t_A)$ 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 that every concrete derivation starting from ρ_C, e is safely approximated by every abstract derivation starting from ρ_A, e. This follows from the proof that $S_0 \subseteq F(S_0)$, where $S_0 = \{(t, t') \mid t \in wftree_C, t' \in wftree_A, root(t') = \rho_A \vdash e \Downarrow a$, and $((root(t) = \rho_C \vdash e \Downarrow c, \rho_C\ safe_{Env}\ \rho_A, c\ safe_{Val}\ a)$ or $(root(t) = \rho_C \vdash e \Uparrow, \rho_C\ safe_{Env}\ \rho_A))\}$.

The proof goes as follows: Consider a pair, $(t, t') \in S_0$; If $t \in wftree_C^+$, then we appeal to the earlier safety result for finite trees. If $t \in wftree_C^-$, then we must prove: *(i)* $root(t)\ safe_{Seq}\ root(t')$—since $root(t)$ 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 explicitly by inspection of the inference rules used to derive t and t', respectively; for our example language, 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 ρ'_C $safe_{Env}$ ρ'_A. Hence, $(t_1, t'_1) \in S_0$. Similarly, $root(t_2) = \rho_C \vdash e_2 \Downarrow c'$, a finite tree, implying that $root(t'_2) = \rho_A \vdash e_2 \Downarrow a'$, c' $safe_{Val}$ a', and therefore $(t_2, t'_2) \in S_0$. Finally, for $root(t_3) = \rho'_C \oplus \{f' \mapsto \langle \rho'_C, f', x', e' \rangle, x' \mapsto c'\} \vdash e' \Uparrow$, we deduce from our knowledge about $root(t'_1)$ and $root(t'_2)$ that $root(t'_3) = \rho'_A \oplus \{f' \mapsto \langle \rho'_A, f', x', e' \rangle, x' \mapsto a'\} \vdash e' \Downarrow a$, for some $a \in AbsVal$. This implies $(t_3, t'_3) \in S_0$.

It is crucial to 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. [58]).

3.4 Termination

We require that an *a.i.* terminate, and memoization can be utilized when the $AbsVal$ set is infinite. Memoization proceeds as in the case of the flowcharts seen in Section 2.6: an *a.i.* is generated in stages, starting from a root sequent, $t_0 = \rho_A \vdash p \Downarrow \perp$. (The $\perp$ means that the synthesized attribute is unknown.) At stage $i+1$, each leaf in t_i are expanded by an inference rule; if $\rho \vdash e \Downarrow \perp$ is a newly generated leaf, the leaf is revised to $\rho \sqcup \rho' \vdash e \Downarrow \perp$, where $\rho' = \sqcup\{\rho'' \mid \rho'' \vdash e \Downarrow a$ appears in $t_i\}$.[12] This gives t_{i+1}. The completed tree is $lim_{i \geq 0} t_i$.

[12]The intuition is that the t_is are generated by a breadth-first Prolog interpreter executing the inference rules.

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*,[13] 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.[14]

3.5 Application: Set-Based Analyses

The $\top$-value and the $\sqcup$-operation that appear in the abstract semantics destroy precision; we prefer to say $f_{\texttt{div2}A}(e)$ equals $\{e, o\}$ 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 instead [27, 34].

When working with sets, the abstract semantics rules use synthesized attributes of the form $\mathcal{P}(AbsVal)$; three of the modified rules from Figure 9 are

$$\frac{\{\rho \vdash e_i \Downarrow S_i\}_{i \in 1..n}}{\rho \vdash op(e_i)_{i \in 1..n} \Downarrow \{f_{opA}(a_i)_{i \in 1..n} \mid a_i \in S_i, i \in 1..n\}}$$

$$\frac{\rho \vdash e_1 \Downarrow S_1 \quad \rho \vdash e_2 \Downarrow S_2 \quad \{\rho_i + (f_i \mapsto cl_i) + (x_i \mapsto v_j) \vdash e_i \Downarrow S_{ij} \mid cl_i = \langle \rho_i, f_i, x_i, e_i \rangle \in S_1, v_j \in S_2\}}{\rho \vdash e_1\ e_2 \Downarrow \cup\{S_{ij} \mid cl_i \in S_1, v_j \in S_2\}}$$

$$\frac{\rho \vdash e_1 \Downarrow S_1 \quad \{\rho \vdash e_i \Downarrow S_i \mid (tt \in S_1 \Rightarrow i = 2), (ff \in S_1 \Rightarrow i = 3)\}}{\rho \vdash \texttt{if e1 e2 e3} \Downarrow \cup\{S_i \mid (tt \in S_1 \Rightarrow i = 2), (ff \in S_1 \Rightarrow i = 3)\}}$$

For example, the first rule states that the evaluation of each argument, e_i, yields a set of values, S_i; the result of $op(e_i)$ must therefore be the set of all $f_A(a_i)_{i \in 1..n}$, for all combinations of $a_i \in S_i$, $i \in 1..n$.

The *a.i.* of the example in Figure 10 is reworked in Figure 12; the precision of the trace increases yet safety is preserved. Notice also that the recursion, $a = \{o\} \cup a$, can be solved in the complete lattice $\mathcal{P}(AbsVal)$ (the computational ordering), giving $a = \{o\}$—there is no need for $\top$ and no need for an approximation ordering upon $AbsVal$.

When sets of closures appear in an *a.i.*, the analysis is called a *closure analysis* [29, 30, 50, 53, 60, 61]. The 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, ℓ : `rec f x.e`. Next, we ignore primitive values in this discussion and define

$$\begin{aligned} AbsVal &= \{\bullet\} \cup AbsClos \\ AbsClos &= AbsEnv \times Label \\ AbsEnv &= Id \xrightarrow{fin} AbsVal \end{aligned}$$

[13] This term is due to Neil Jones.

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

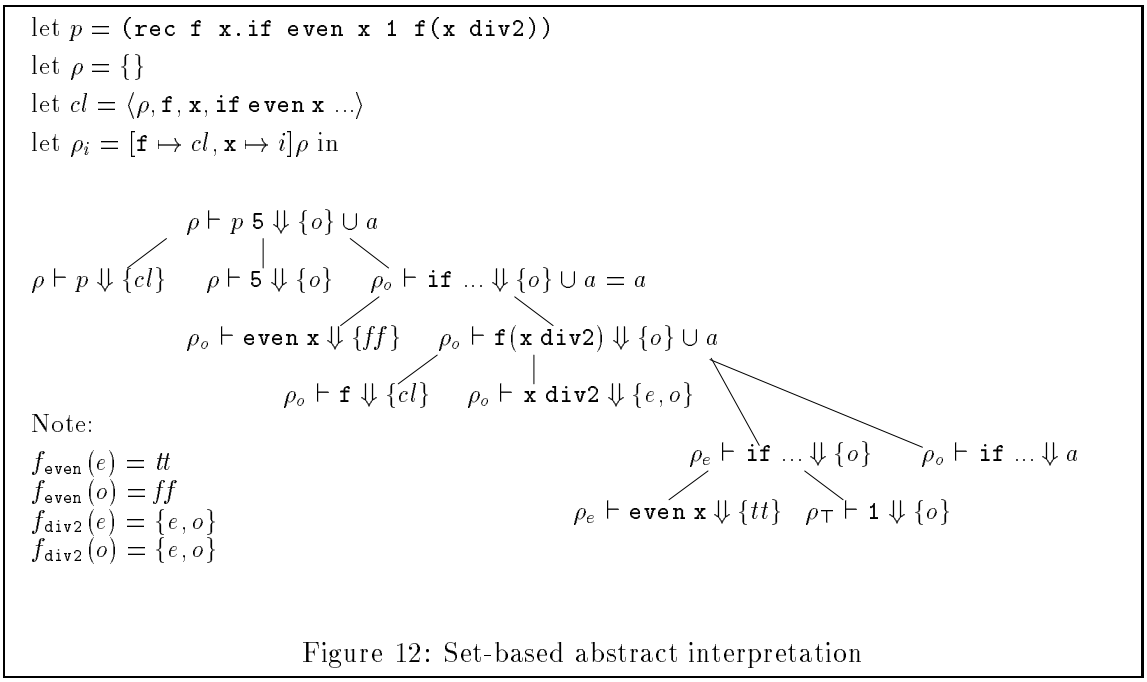


Figure 12: Set-based abstract interpretation

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

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

Now, the task is finding simplifications of *AbsClos*. One that is related to the n-CFA analyses proposed by Shivers [50, 61] defines *AbsClos* as $AbsClos_n$, for some fixed $n \geq 0$, where

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

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 $n = 0$ [60, 61], but a complication to limiting the depth of closures is that a function application might be forced to synthesize an environment for the closure $\ell \mapsto \{\bullet\}$ when the closure is applied to an argument. (Indeed, this is *always* the case for 0-CFA.) There are a variety of safe methods for synthesizing the environment—a natural one examines the derivation tree for sequents of the form $\rho' \vdash \ell : \mathtt{rec\ f\ x.e} \Downarrow \{\ell \mapsto \{\bullet\}\}$ and joins the respective ρ's, thus (safely) confusing the creation sites (cf. "call sites" [62]) of the closure.

4 Small-step semantics

Flowchart semantics is an example of a small-step semantics, so called because it rewrites a program configuration, step by step, to a final or normal form. Here, we examine the

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

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

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

Figure 13: Subset of CCS-like language

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

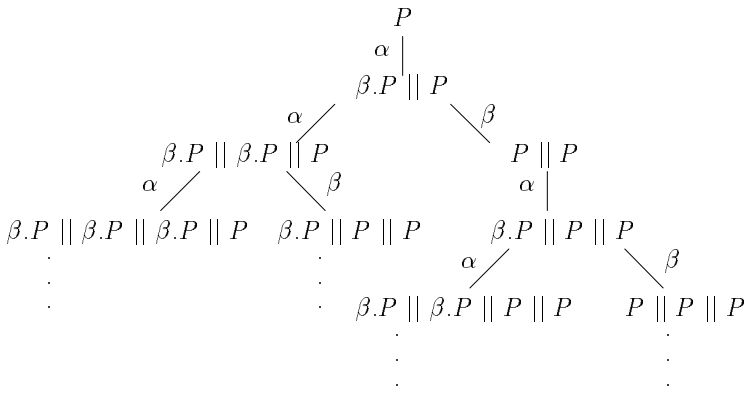


Figure 14: Example of process creation

more general Plotkin-style structural operational semantics (*SOS*) [51, 55], where state transitions are defined by inference rules. The techniques developed for flowchart analyses by and large adapt, but a new problem arises because SOS definitions can generate new program syntax on the fly—unlike flowchart semantics, arbitrary SOS derivations are not semicompositional; this can hinder the termination of an *a.i.*

As an example, Figure 13 shows a CCS-like notation [41] with a small-step semantics that spawns processes via unfolding of a recursion constructor, μ.[15] Because the μ-rule generates new program syntax, the semantics derivations will not be semicompositional. Figure 14 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" [48, 49]. For example, the configuration $\beta.P \,||\, \beta.P \,||\, P$ is written as the bag $\{\beta.P, \beta.P, P\}$.

[15] 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: $\dfrac{[\mu p(e)/p]e \xrightarrow{\alpha} e'}{\mu p(e) \xrightarrow{\alpha} e'}$, which operates on closed terms only but allows unbounded unfolding. Nonetheless, the rules generate bisimular behavior trees.

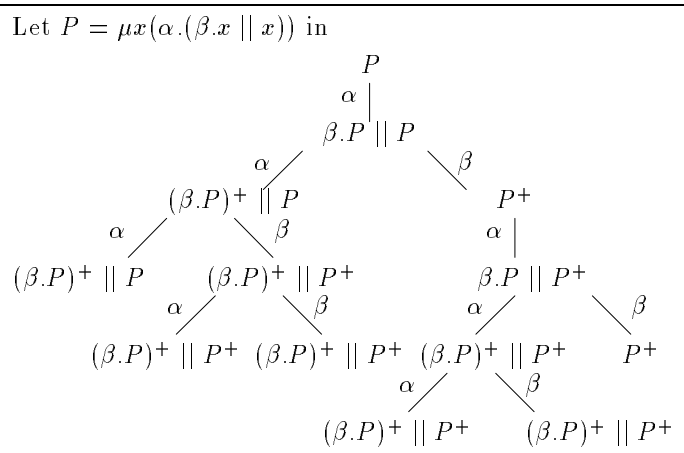


Figure 15: Abstract interpretation of process creation

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 is encoded by the function $[\ell_3 \mapsto 2][\ell_0 \mapsto 1](\lambda\ell.0)$.[16]

Our task is to abstractly represent the process pools, which we do by defining the abstract pools to be functions of the form $Label \to \{0, 1, \omega\}$. 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 $cp \in Pool$, $ap \in AbsPool$,

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

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

Figure 15 shows the abstract interpretation of the derivation in Figure 14. For simplicity, an abstract pool is written as a regular expression, where a "+" marks a process whose count is ω. Using this representation of the abstract pools, we see that the semantics rules in Figure 13 can be used as the abstract semantics rules along with one new rule which accounts for processes whose count is ω:

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

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

Of course, the purpose of abstractly interpreting the syntax is to restore a form of semi-compositionality. The regular expressions used here were first devised by Codish, Falaschi

[16] 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.

and Marriott as "*-abstractions" [9]. 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 [22].

4.1 Safety and Liveness Properties

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

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

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

$$\begin{array}{c} cp\ safe_{Pool}\ ap \text{ and } cp \stackrel{\alpha}{\rightarrow} cp' \text{ imply there exists } ap' \\ \text{such that } ap \stackrel{\alpha}{\rightarrow} ap' \text{ and } cp'\ rel_{Pool}\ ap' \end{array}$$

Recall that a safe *a.i.* can be model checked consistently with the box-mu-calculus—any property that holds true of the *a.i.* holds for the corresponding *c.i.* s. As before, liveness is the dual relation: the *c.i.* is a simulation of the *a.i.* :

$$\begin{array}{l} t\ live_{Trace}\ u \text{ iff } root(t)\ live_{Pool}\ root(q) \\ \quad \text{and for every transition}, root(u) \stackrel{\alpha}{\longrightarrow} u_i, \text{ there exists a transition}, \\ \quad root(t) \stackrel{\alpha}{\longrightarrow} t_j, \text{ such that } t_j\ live_{Trace}\ u_i \end{array}$$

For the example in this section, a liveness *a.i.* requires a different semantics than the safety *a.i.* : an abstract pool must be a mapping in $Label \rightarrow \{0, 1, \ldots, n\}$, for some fixed positive n. For example, for $n = 2$, the abstract pool representation of $\beta.0 \mid\mid \beta.0 \mid\mid \alpha.0 \mid\mid \beta.0$ is $(\beta.0)^2 \mid\mid \alpha.0$.

The relation between concrete pools and the new abstract pools is

$$cp\ live_{Pool}\ ap \text{ iff for all } \ell \in Label, cp(\ell) \geq ap(\ell)$$

and the abstract semantics rules are the concrete rules along with

$$\frac{e \stackrel{\alpha}{\rightarrow} e'}{e^n \stackrel{\alpha}{\rightarrow} e^{n-1} \mid\mid e'} \text{ for } n > 0, \text{ where } e^1 \equiv e \text{ and } e^0 \equiv 0$$

Recall that a live *a.i.* can be model checked consistently with the diamond-mu-calculus.

As noted by Dams [17], ideally the abstract pools for safety and liveness analyses should be the same, because this lets one model check the full μ-calculus.

4.2 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 syntax is altered to read

$$e \ ::= \ 0 \ \mid \ \alpha?x.e \ \mid \ \alpha!\alpha'.e \ \mid \ e_1 \mid\mid e_2 \ \mid \ \mu p(e) \ \mid \ p$$

$c \in$ 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

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

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:

$$\rho \vdash \alpha?x.e \xrightarrow{\alpha?\alpha'} \rho \oplus \{x = \alpha'\} \vdash e \qquad \rho \vdash \alpha!\alpha'.e \xrightarrow{\alpha!\alpha'} \rho \vdash e \qquad \frac{\rho \vdash e \xrightarrow{\Delta} \rho' \vdash e'}{\rho \vdash \mu p(e) \xrightarrow{\Delta} \rho' \oplus \{p \mapsto \rho\} \vdash e'}$$

$$\frac{c_1 \xrightarrow{\alpha?\alpha'} c_1' \quad c_2 \xrightarrow{\alpha!\alpha'} c_2'}{c_1 \,||\, c_2 \xrightarrow{\tau} c_1' \,||\, c_2'} \qquad \frac{c_1 \xrightarrow{\Delta} c_1'}{c_1 \,||\, c_2 \xrightarrow{\Delta} c_1' \,||\, c_2} \qquad \frac{c_2 \xrightarrow{\Delta} c_2'}{c_1 \,||\, c_2 \xrightarrow{\Delta} c_1 \,||\, c_2'}$$

Figure 16: Concrete small-step semantics of channel passing

where $\alpha?x.e$ inputs a value on channel α and binds it to identifier x within scope e, and $\alpha!\alpha'.e$ outputs channel α' on channel α and proceeds with e. The relevant semantics rules are

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

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

This semantics defines argument transmission via substitution, but substitutions generate new program syntax, which hampers 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 $\rho \vdash e$. Although it is not essential, we eliminate the substitution semantics for $\mu p(e)$ by saving a "closure" of $\mu p(e)$ in the environment of the unfolded process, e. Figure 16 shows the modifications.

To simplify matters, we assume that argument identifiers are distinct from process indentifiers; also, assume that each recursive process, $\mu p(e)$, uses a unique process identifier, p. Environments hold bindings of argument identifiers to channels, $x_i = \alpha_i$, and bindings of process identifiers to "closures," $p_j \mapsto \rho_j$.

The computation rules are kept simple by employing the usual symmetric, monoidal congruences plus three more: the first additional congruence explains how an environment

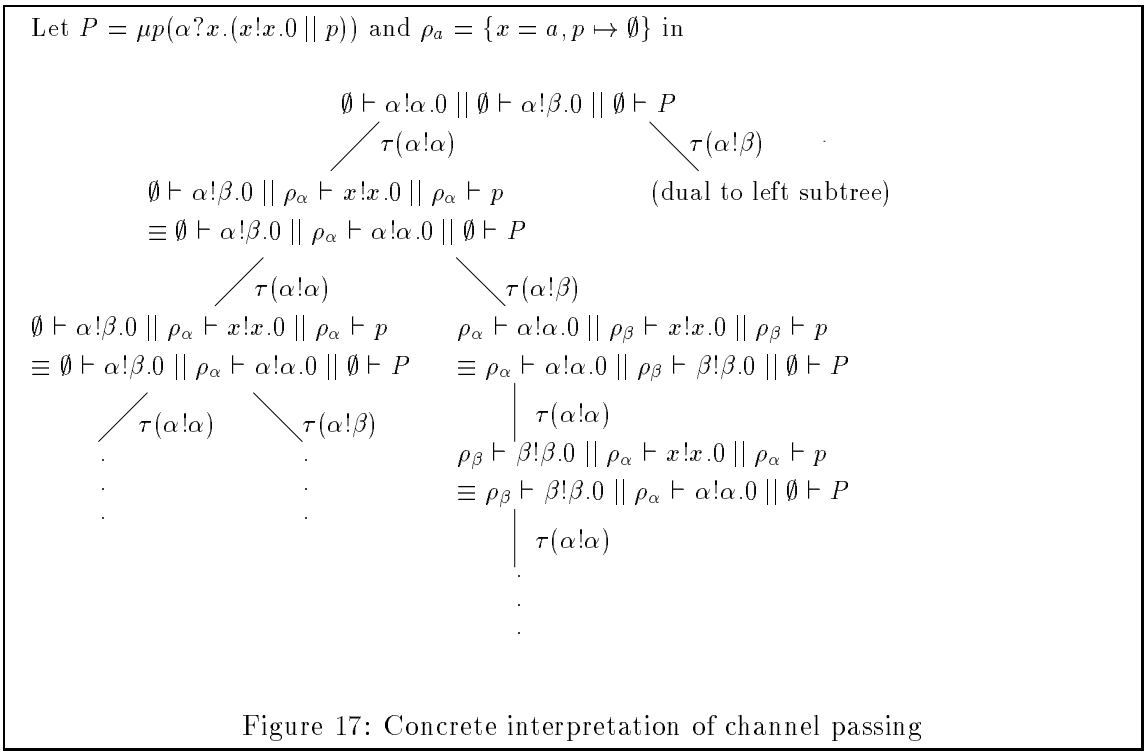


Figure 17: Concrete interpretation of channel passing

is copied to component processes of a system; the second shows how identifier lookup in the environment is employed; and the third describes recursive process unfolding. 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 evaluated. Figure 17 displays an example *c.i.*

Now, a program's process pool representation is a bag of (environment,process) pairs. Assuming that processes are labelled, a process pool is concisely depicted as a function of form $Label \rightarrow Bag(Environment)$, where $Environment = (ArgumentId \rightarrow Channel) \times (ProcessId \rightarrow Environment)$.[17]

A manageable abstract semantics defines an abstract pool to be a function of the form $Label \rightarrow AbsEnv \times \{0, 1, \omega\}$, where $AbsEnv = (ArgumentId \rightarrow \mathcal{P}(Channel)) \times (ProcessId \rightarrow AbsEnv)$. That is, an abstract environment associates an argument identifier with a set of possible channels that the identifier might denote.

The safety relation between concrete and abstract process pools becomes

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

where concrete and abstract environments are related as follows:

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

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 $\{x = \{\alpha, \beta\}\} \vdash (x!x.0)^+ \,||\, \emptyset \vdash \alpha?y.0$.

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

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

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

The abstract semantics rules are complicated by the sets of channels: If we use the congruence rule in Figure 16 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:

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

The entire set of channels, S', is transmitted along any $\alpha \in S$. This is a form of "independent attribute" analysis of channel flow [33].

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, $\{x = \{\alpha, \beta\}\} \vdash x!x.0 \equiv (\{x = \{\alpha, \beta\}\} \vdash \alpha!\alpha.0) + (\{x = \{\alpha, \beta\}\} \vdash \beta!\beta.0)$.[18] 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. Of course, the price paid for the extra precision of the relational analysis is a slower convergence of construction of the regular tree.

Figure 18 shows the regular tree that results from the relational analysis of the program in Figure 17. 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 $\alpha!\alpha$, 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).

5 Conclusion

The examples in this paper demonstrate that the principles of abstract interpretation apply simply and uniformly to the members of the operational semantics hierarchy. In particular, the trace-based representations of concrete and abstract interpretations make clear that not only does *a.i.* apply uniformly across the hierarchy but there exist close connections between *a.i.* and methodologies for flow analysis, liveness validation, and model checking. Unifying these areas within a general framework must be the next challenge.

[18] 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.

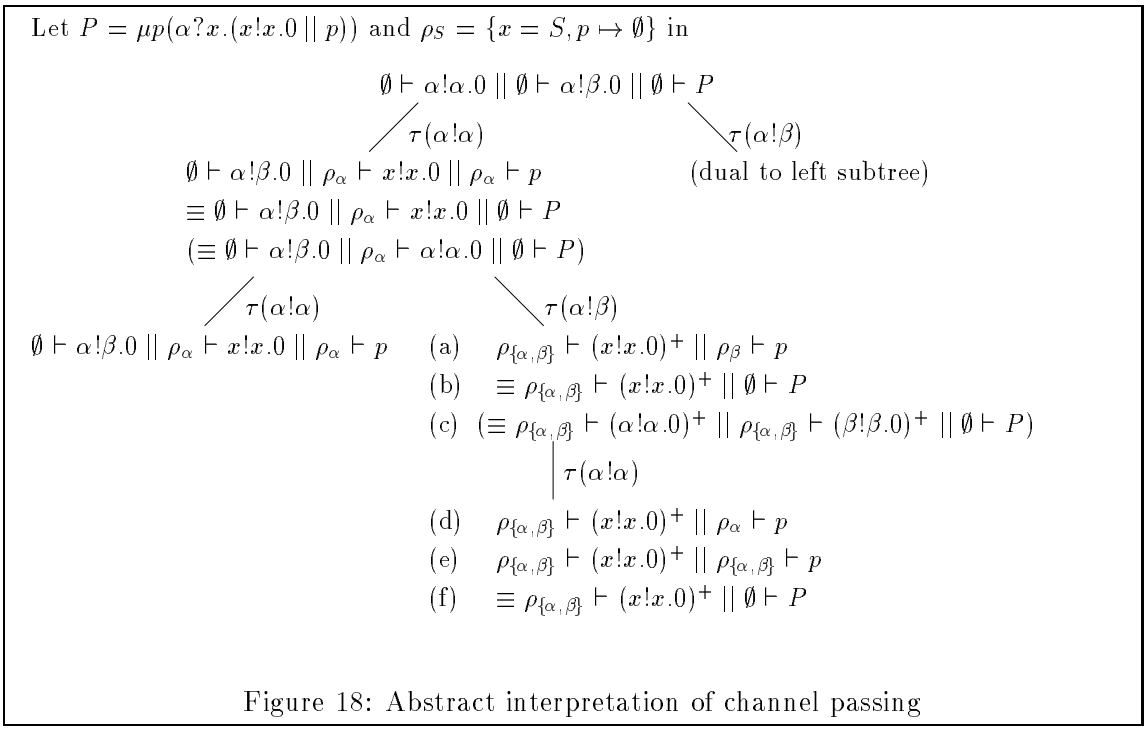


Figure 18: Abstract interpretation of channel passing

6 Acknowledgements

Mitchell Wand studied an earlier draft of this paper and made many useful suggestions. Helpful comments and assistance were provided by Ed Clarke, Radhia and Patrick Cousot, Chris Hankin, Daniel Jackson, Thomas Jensen, Daniel LeMetayer, Flemming and Hanne Nielson, Frank Pfenning, David Sands, and Andrew Wright.

References

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

[2] P. Aczel. *Non-Well-Founded Sets.* Lecture Notes 14, Center for Study of Language and Information, Stanford, CA, 1988.

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

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

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

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

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

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

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

[39] 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.

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

[41] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

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

[43] 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.

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

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

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

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

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

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

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

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

[52] 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.

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

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

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

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

[57] 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.

[58] 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.

[59] 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.

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

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

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

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

[64] 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.

[65] 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.

[66] 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.

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

Recent BRICS Report Series Publications

RS-97-2 David A. Schmidt. *Abstract Interpretation in the Operational Semantics Hierarchy*. March 1997. 33 pp.

RS-97-1 Olivier Danvy and Mayer Goldberg. *Partial Evaluation of the Euclidian Algorithm (Extended Version)*. January 1997. 16 pp. To appear in the journal *Lisp and Symbolic Computation*.

RS-96-62 P. S. Thiagarajan and Igor Walukiewicz. *An Expressively Complete Linear Time Temporal Logic for Mazurkiewicz Traces*. December 1996. i+13 pp. To appear in *Twelfth Annual IEEE Symposium on Logic in Computer Science*, LICS '97 Proceedings.

RS-96-61 Sergei Soloviev. *Proof of a Conjecture of S. Mac Lane*. December 1996. 53 pp. Extended abstract appears in Pitt, Rydeheard and Johnstone, editors, *Category Theory and Computer Science: 6th International Conference*, CTCS '95 Proceedings, LNCS 953, 1995, pages 59–80.

RS-96-60 Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL *in 1995*. December 1996. 5 pp. Appears in Margaria and Steffen, editors, *Tools and Algorithms for The Construction and Analysis of Systems: 2nd International Workshop*, TACAS '96 Proceedings, LNCS 1055, 1996, pages 431–434.

RS-96-59 Kim G. Larsen, Paul Pettersson, and Wang Yi. *Compositional and Symbolic Model-Checking of Real-Time Systems*. December 1996. 12 pp. Appears in *16th IEEE Real-Time Systems Symposium*, RTSS '95 Proceedings, 1995.

RS-96-58 Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL *— a Tool Suite for Automatic Verification of Real–Time Systems*. December 1996. 12 pp. Appears in Alur, Henzinger and Sontag, editors, *DIMACS Workshop on Verification and Control of Hybrid Systems*, HYBRID '96 Proceedings, LNCS 1066, 1996, pages 232–243.

RS-96-57 Kim G. Larsen, Paul Pettersson, and Wang Yi. *Diagnostic Model-Checking for Real-Time Systems*. December 1996. 12 pp. Appears in Alur, Henzinger and Sontag, editors, *DIMACS Workshop on Verification and Control of Hybrid Systems*, HYBRID '96 Proceedings, LNCS 1066, 1996, pages 575–586.