Abstract Interpretation of Small-Step Semantics

David A. Schmidt
Kansas State University[1]

Abstract. The techniques of classical abstract interpretation are extended to the big- and small-step operational semantics of higher-order and communicative languages: Well-known techniques, such as memoization, and lesser-known ones, such as abstraction on program syntax, are employed to generate finite abstract interpretations of source programs based on their formal operational semantic definitions. The result is a clear methodology for generating semantically safe (and live) abstract, regular trees for programs that do not possess obvious, finite, state-transition diagram depictions. The primary application of the research is to the validation of program properties; in particular, the application of model checking to validate safety properties in the box-mu calculus and liveness properties in the diamond-mu calculus is discussed.

1 Introduction

Two motivations triggered the research in this paper: *(i)* A desire to generalize the elegant abstract intepretation techniques of Cousot and Cousot [8] from flowchart languages to higher-order and communicative languages by abstractly interpreting big-step and small-step operational semantics; *(ii)* Application of the results of (i) to model checking infinite-state systems for safety and liveness properties.

The emphasis of this paper lies upon Topic (i), since it is crucial that semantically correct abstract interpretations exist before analysis techniques like model checking are employed. Nonetheless, a discussion of Topic (ii) and related research is presented once (i) has been achieved.

For Topic (i), the abstract interpretation methodology is an extension of earlier research on abstract interpretation of big-step (natural) semantics [25]. As we will see, a distinguishing feature of small-step semantics is the generation of new program syntax during the calculation of a source program's semantics. (This can happen when a concurrent program spawns new processes, for example.) This complicates convergence of the abstract interpretation but can be handled by abstractly interpreting the source language syntax itself, using techniques from analysis of logic programs [5, 6].

For Topic (ii), we see that abstract interpretation techniques are ideally suited for safety checking, where the box-mu-calculus (that is, the modal-mu-calculus less the diamond modality [30]), can be used as the safety specification language. Liveness checking of properties encoded in the diamond-mu-calculus can also be performed on an abstract interpretation, but the abstractions used for liveness

[1] Computing and Information Sciences Department, 234 Nichols Hall, Manhattan, KS 66506 USA. `schmidt@cis.ksu.edu`. Supported by NSF CCR-9302962 and ONR N00014-94-1-0866.

tend to be different than those used for safety, and this presents challenges for practical application. Some suggestions for future research will be made.

The paper is organized as follows: a review of the abstract interpretation methodology for big-step semantics comes first, then an adaption of the methodology for small-step semantics follows. Next, the methodology is applied to a CCS-subset and a π-calculus subset. Safety and liveness issues are discussed. Finally, related work is examined and future developments are suggested.

2 Analysis of Big-Step Semantics

The classic example of safety checking is the calculation of the input domains of program points by a data-flow analyzer. In abstract interpretation theory, the input domain sets for a program's phrases is called the program's *collecting semantics*. Property checking depends crucially on the collecting semantics, so we focus initially upon it.

We begin with abstract interpretation of big-step (natural) semantics. The methodology proceeds in three stages:

- Given a concrete, environment-based semantics,[2] define the concrete operational semantics of a program as the (possibly infinite) derivation tree drawn with the semantics rules; next, define the program's concrete collecting semantics as the set of (environment, program subphrase) pairs attached to the nodes of the tree.
- Replace the concrete value sets by abstract value sets and define the abstract operational semantics of the program as the (probably infinite) derivation tree drawn with the semantics rules. Define the abstract collecting semantics as the collecting semantics of the tree.
- Finitely approximate the program's abstract operational semantics as a regular tree; define the data-flow analysis of the program as the collecting semantics of the regular tree.

Since the technical details are available in [10, 25], the methodology is presented here via an example.

Figure 1 shows two of the rules that might be found in a big-step semantics of a functional programming language with recursive definitions, and Figure 2 shows the concrete, operational semantics of the program `(rec f x.f(x+1)+1)2`, which is an infinite derivation tree. In the derivation, a box at a node represents an unknown value to be supplied by a result from a child node; as noted in [25], a least-fixed point interpretation of derivation tree construction would insert $\perp$ into all boxes in the figure.

[2] Identifier bindings are represented by an environment rather than by syntactic substitution.

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

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

Figure 1: Sample big-step semantics rules

Let $\rho_i = \rho \oplus \{f = cl\} \oplus \{x = i\}$ and $cl = \mu_f\langle \texttt{x},\ \texttt{f(x+1)+1},\ \rho\rangle$ in

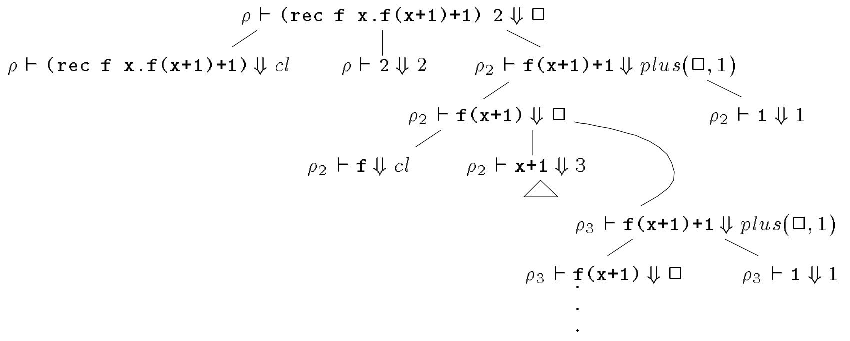

Figure 2: Concrete semantics derivation

The concrete collecting semantics of the program is depicted as follows:

Subphrase	*Environments*
`(rec f x.f(x+1)+1) 2`	$\{\rho\}$
`rec f x.f(x+1)+1`	$\{\rho\}$
`2`	$\{\rho\}$
`f(x+1)+1`	$\{\rho_i \mid i \geq 2\}$
`f`	$\{\rho_i \mid i \geq 2\}$
...	...

Next, say that the program is to be abstractly interpreted for sign properties: the concrete value set of integers, *Int*, is replaced by the set $\{neg, zero, pos\}_\perp^\top$; the semantics rules are revised to calculate upon sign properties, most notably,

$$\frac{\rho \vdash e_1 \Downarrow v_1 \quad \rho \vdash e_2 \Downarrow v_2}{\rho \vdash e_1 + e_2 \Downarrow plus(v_1, v_2)} \quad \text{becomes} \quad \frac{\alpha \vdash e_1 \Downarrow a_1 \quad \alpha \vdash e_2 \Downarrow a_2}{\alpha \vdash e_1 + e_2 \Downarrow absplus(a_1, a_2)}$$

where α is the abstract environment argument, which maps identifiers to signs, and *absplus* is addition on signs (e.g., $absplus(pos, pos) = pos$, $absplus(pos, neg) = \top$, etc.). The abstract semantics is calculated as the infinite derivation tree in Figure 3, and the abstract collecting semantics goes as follows:

Let $\alpha_{pos} = \rho \oplus \{f = cl\} \oplus \{x = pos\}$ and $cl = \mu_f\langle$`x, f(x+1)+1,` $\alpha\rangle$ in

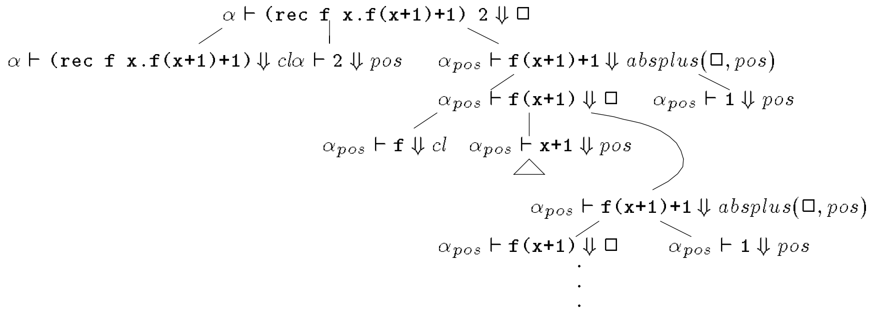

Figure 3: Abstract semantics derivation

Subphrase	*Environments*
`(rec f x.f(x+1)+1) 2`	$\{\alpha\}$
...	...
`f(x+1)+1`	$\{\alpha_{pos}\}$
`f`	$\{\alpha_{pos}\}$
...	...

Of course, a data flow analysis does not have the luxury of drawing the infinite abstract tree, so one must ensure that one can always draw a finitely presented tree that contains the collecting information in the infinite derivation. In the example in Figure 3, one notes immediately that the tree is *regular*, that is, every infinite path contains a node that repeats. Therefore, one can terminate the unfolding of the infinite path at the "repetition node." In the example, the repetition node is $\alpha_{pos} \vdash$ `f(x+1)+1` $\Downarrow$ $absplus(\Box, pos)$. Indeed, once the repetition node appears, one can calculate the values that should be inserted into the boxes in the tree by a least-fixed-point calculation: whatever value, v, should appear at the repeating node, $\alpha_{pos} \vdash$ `f(x+1)+1` $\Downarrow v$, it must be the case that $v = absplus(v, pos)$. The least such value that satisfies this equality is $\bot$, so $\bot$ fills the boxes in the figure.

In the general case, one is not guaranteed that an abstract derivation tree will be regular—sometimes one must "force" it to happen. As an example, consider a constant propagation analysis of the program in Figure 2, where the integer value set, *Int*, is replaced by $Int_{\bot}^{\top}$; the "abstract" derivation of the program looks exactly like Figure 2, and it is not a regular tree.

To force the issue, we apply the well-known technique from iterative data-flow analysis of memoizing the environments that appear with the subphrases of the program at the nodes of the derivation tree. When a subphrase, e, repeats in a path of the tree, the values of the environments that appeared with the earlier occurrences of e are joined to the current value of the environment that

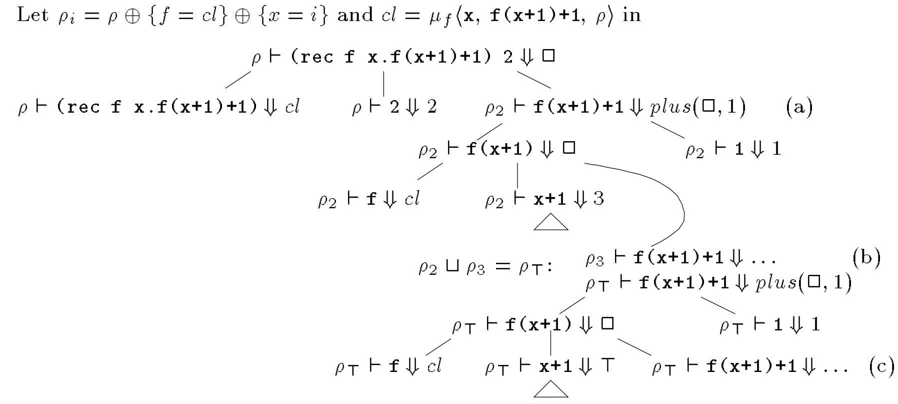

Figure 4: Abstract, regular tree derivation

appears with the repeated occurrence of e.[3] Because of the finite chain property of the abstract value sets, this forces a repetition node to appear in the path, and a regular tree results.

Figure 4 shows this technique applied to the constant propagation analysis of the program in Figure 2. The syntax phrase `f(x+1)+1` at node (a) repeats at (b), so the environment ρ_2 at (a) is joined to the environment ρ_3 at (b), giving $\rho_2 \sqcup \rho_3 = \rho_\top$, which is used to continue the derivation. This quickly produces a repetition node at (c). Now, we can determine the values that should be inserted into the boxes in the tree. The value to calculate first is the one for node (c): since node (c) is a repetition of node (b), we must find the least value, v, such that $v = plus(v, 1)$ holds. The answer is $\bot$, and substituting $\bot$ for the box at node (c) fills the rest of the boxes with $\bot$ also.

The collecting semantics of the regular tree is

Subphrase	*Environments*
`(rec f x.f(x+1)+1) 2`	ρ
...	...
`f(x+1)+1`	$\rho_\top$
`f`	$\rho_\top$
...	...

This is a safe approximation of the abstract collecting semantics and is the information calculated by a classic iterative data-flow analysis.

A crucial feature of the environment-based, big-step semantics that ensures the construction of regular trees is that the semantics is *semicompositional*,[4] that is, all syntax phrases that are analyzed in the calculation of the program's semantics are subphrases of the original source program. Thus, there are at

[3] This technique can be formalized as a *widening* of the least-fixed point calculation of the abstract semantics tree[8].

[4] This term is due to Neil Jones.

most a finite number of subphrases that can appear at nodes of the derivation tree.[5] This, plus the finite chain property of the abstract value sets, ensures that regular trees must result from the memoization construction.

The correctness criterion for the methodology is straightforward: To relate the concrete semantics derivation tree to the abstract tree, define a safety relation rel_{Tree} as follows: for concrete tree, t, and abstract tree, u,

$$
\begin{array}{l}
t\ rel_{Tree}\ u \text{ iff } root(t)\ rel_{Node}\ root(q) \\
\quad \text{and for every subtree, } t_i, \text{ there exists a subtree, } u_j, \text{ such that } t_i\ rel_{Tree}\ u_j \\
\text{where } (\rho \vdash e \Downarrow v)\ rel_{Node}\ (\alpha \vdash e \Downarrow a) \text{ iff } \rho\ rel_{Env}\ \alpha \text{ and } v\ rel_{Value}\ a \\
\text{and } \rho\ rel_{Env}\ \alpha \text{ iff } domain(\rho) = domain(\alpha) \\
\qquad\qquad \text{and for all } i \in domain(\rho), \rho(i)\ rel_{Value}\ \alpha(i)
\end{array}
$$

The definition of rel_{Value} depends on the particular application area, e.g., for sign analysis, $-n\ rel_{Value}\ neg$, $+n\ rel_{Value}\ pos$, etc.[6]

The safety relation rel_{Tree} states that every path in the concrete derivation tree is approximated by a path in the abstract derivation tree. Since rel_{Tree} is defined recursively and infinite trees are involved, one wants the largest such relation satisfying the definition (that is, the greatest-fixed-point interpretation), and the related coinductive proof techniques prove the main safety result, namely: for all expressions, e, and environments ρ and α such that $\rho\ rel_{Env}\ \alpha$, $t\ rel_{Tree}\ u$ must hold, where t is the concrete derivation tree whose root is $\rho \vdash e \Downarrow v$ and u is the abstract derivation tree with root $\alpha \vdash e \Downarrow a$. A corollary of the safety result is that the collecting semantics of the abstract tree safely approximates the collecting semantics of the concrete tree.

Once the safety of the abstract interpretation is established, one notes that the regular tree built with the memoization technique safely approximates the abstract semantics derivation tree.[7] This result implies that the concrete tree is safely approximated by the regular tree.[8]

Of course, the pragmatics of data-flow analysis demands an implementation technique that is more efficient than the regular-tree-building methodology described in this section. Nonetheless, the methodology is important because it produces simple correctness proofs [10, 25] and separates design concerns from implementation concerns. For this reason, we pursue its application to small-step semantics.

---

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

[6] For closure values, there are two possibilities for rel_{Value} : the simpler definition is $\langle x, e, \rho\rangle\ rel_{Value}\ \langle x, e, \alpha\rangle$ iff $\rho\ rel_{Env}\ \alpha$. In a simply typed language, a more complex definition can be helpful: $\langle x, e, \rho\rangle\ rel_{Value}\ \langle x', e', \alpha\rangle$ iff for all v, a such that $v\ rel_{Value}\ a$, $apply_{con}(\langle x, e, \rho\rangle, v)\ rel_{Tree}\ apply_{abs}(\langle x', e', \alpha\rangle, a)$, where $apply_{con}$ and $apply_{abs}$ are the tree-building operations for concrete and abstract function application, respectively.

[7] This is formalized like above: for abstract tree, u, and regular tree, r, $u\ rel_{Atree}\ r$ iff $root(u)\ rel_{Anode}\ root(r)$ and for every subtree, u_i, there exists a subtree r_j, such that $u_i\ rel_{Atree}\ r_j$. Next, $(\alpha \vdash e \Downarrow a)\ rel_{Anode}\ (\alpha' \vdash e \Downarrow a')$ iff $\alpha \sqsubseteq \alpha'$ and $a \sqsubseteq a'$.

[8] We must require, however, that rel_{Value} is *U-closed*, that is, $v\ rel_{Value}\ a_1$ and $a_1 \sqsubseteq a_2$ imply $v\ rel_{Value}\ a_2$ [25].

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

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

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

Figure 5: Subset of CCS-like language

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

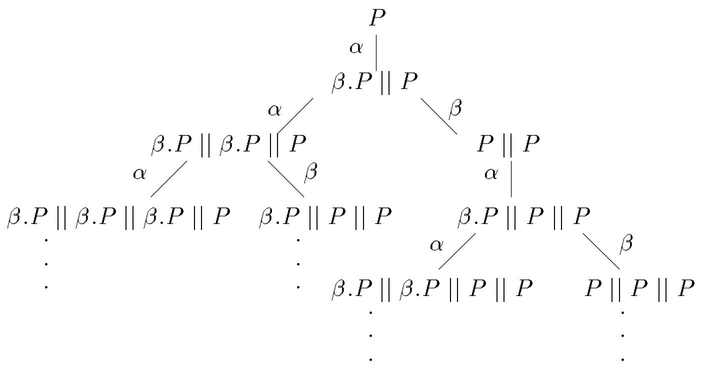

Figure 6: Example operational semantics

3 A CCS Subset and its Analysis

A Plotkin-style structural operational semantics [23] and a term rewriting system [18] are two examples of small-step semantics definitions—both rewrite a program configuration, step by step, to an answer or normal form. If a configuration contains multiple redexes so that there exist multiple possible next stages, we will draw the rewriting as a *behavior tree*, with one path for each of the contracted redexes. Figure 5 shows a CCS-like notation [20] whose small-step semantics spawns processes via unfolding of a recursion constructor, μ.[9] Because the μ-rules generates new program syntax, the semantics derivations will not be semicompositional. Figure 6 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. Clearly, to undertake an abstract interpretation, some means must be found to

[9] The rule for μ in the figure unfolds a μ-process exactly once when it makes a communication step. This differs from the bisimular 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.

approximate finitely the dynamically generated processes.

An abstract interpretation must approximate the program configurations that arise during execution. To do this, we abstractly interpret the source program syntax itself.[10] To set the stage for the abstract interpretation, it is helpful to view a program configuration as a "process pool" [21, 22], that is, as a bag of processes. For example, the configuration $\beta.P \mid\mid \beta.P \mid\mid P$ is written as the bag $\{\beta.P, \beta.P, P\}$.

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

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

Then a process pool is coded by a function of the form $Label \rightarrow Nat$, e.g., the bag just above is encoded by the function $[\ell_3 \mapsto 2][\ell_0 \mapsto 1](\lambda\ell.0)$.[11]

To perform an abstract interpretation, one must define a finite-height, sup-semilattice of abstract process pools. For the current example, we define an abstract process pool to be a function of the form $Label \rightarrow \{0, 1, \omega\}$. Since both domain and codomain of this function space are finite, there exist a finite number of abstract process pools, that is, program configurations. The concrete process pool above is abstracted by $[\ell_3 \mapsto \omega][\ell_0 \mapsto 1](\lambda\ell.0)$, and the logical relationship between concrete and abstract process pools is of course: for $cp \in Pool$, $ap \in AbsPool$,

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

The bag union operation, $\uplus$, is interpreted abstractly in the expected way: $ap_1 \uplus ap_2 = \lambda\ell.ap_1(\ell) \oplus ap_2(\ell)$, where $1 \oplus 1 = \omega$, and $m \oplus n = \max\{m, n\}$ otherwise.[12]

Figure 7 shows the abstract interpretation of the derivation in Figure 6. For simplicity, an abstract process pool is written as a kind of regular expression, where a "+" marks a process whose count is ω. This reveals our abstraction technique to be the "star abstraction" of Codish, Falaschi, and Marriott [6, 5]. Using this representation of abstract process pools, the semantics rules in Figure 5 can be used as the abstract semantics rules along with one new rule which accounts for processes whose count is ω:

$$\frac{e \overset{\alpha}{\rightarrow} e'}{e^+ \overset{\alpha}{\rightarrow} e^+ \mid\mid e'}$$

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

---

[10] Indeed, the purpose of a *closure analysis* [24, 26, 27] of a functional program is to abstractly interpret syntax phrases—lambda abstractions, in particular—that are passed about and evaluated during execution of the program. Both closure analysis and the analysis in this section are "control flow analysis," and additional work is planned to demonstrate the relationship between the two.

[11] 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.

[12] That is, let $[\![e]\!]$ denote e's process pool representation. Then, $[\![e_1 \mid\mid e_2]\!] = [\![e_1]\!] \uplus [\![e_2]\!]$.

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

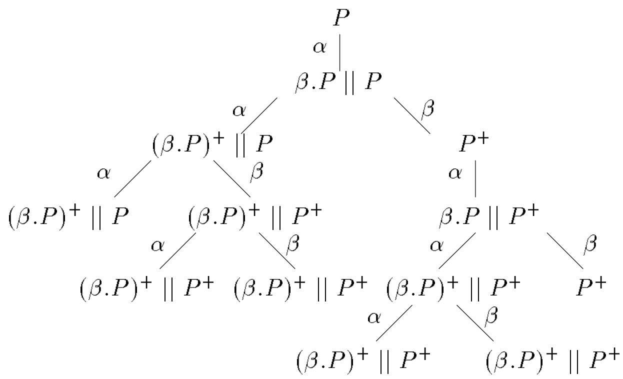

Figure 7: Abstract interpretation of process creation

The safety proof for the abstract interpretation goes the same way as the one for big-step semantics: one must show that every path in the concrete behavior tree is approximated by a path in the abstract behavior tree. This can be proved in terms of a relation, rel_{Tree}, defined like the one in the previous section:

$t \; rel_{Tree} \; u$ iff $root(t) \; rel_{Pool} \; root(q)$
and for every transition, $root(t) \stackrel{\alpha}{\to} t_i$, there exists a transition,
$root(u) \stackrel{\alpha}{\to} u_j$, such that $t_i \; rel_{Tree} \; u_j$

If one can prove the following simulation property

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

then one obtains by coinduction the corollary that the concrete behavior tree is related to the abstract tree. The simulation property is easy to prove for the semantics rules used here.

3.1 Safety Analysis and Liveness Analysis

Once an abstract, regular tree has been constructed, an analysis of the tree can proceed. Compiler code improvement is usually based upon properties of the tree's collecting semantics, but Steffen [28, 29] has noted that collecting-semantics techniques can be encoded as model-checking problems, so we focus on the latter.

Say that the interesting properties of derivation tree nodes are defined by a language, $\mathcal{P}$, of atomic propositions; we write $n \models p$ when property p holds true at node n.

To use the results of an abstract interpretation to check propositions in $\mathcal{P}$, one needs this consistency property for concrete tree nodes, c, and abstract tree nodes. a:

$$c\ rel_{Node}\ a \text{ implies, for all } p \in \mathcal{P},\ \text{ that } a \models p \text{ implies } c \models p$$

Ideally, the first implication in the line above should be an equivalence [9], so that the abstract interpretation computes directly upon propositions, but this is not essential.

The examples developed so far in this paper are oriented towards safety properties: for concrete tree, t, and abstract tree, u, $t\ rel_{Tree}\ u$ iff every computation path in t is safely approximated by a path in u. Thus, one can model check such abstract trees with the box-mu-calculus [30]:

$$\begin{array}{l} p \in \mathcal{P}, \quad \alpha \in \text{Channel}, \quad \phi \in \text{State-formula} \\ \quad \phi \ ::= \ p \ \mid \ \neg p \ \mid \ [\alpha]\phi \ \mid \ \phi_1 \wedge \phi_2 \ \mid \ \phi_1 \vee \phi_2 \ \mid \ \mu x.\phi \ \mid \ \nu x.\phi \end{array}$$

Using the consistency property on atomic propositions, it is easy to prove that the property lifts to the box-mu-calculus: $t\ rel_{Tree}\ u$ and $root(u) \models \phi$ imply $root(t) \models \phi$.

A liveness analysis can be undertaken in a similar fashion: starting from a consistent relation, rel_{Node} , one must generate abstract trees that satisfy this liveness relation:

$$\begin{array}{l} t\ live_{Tree}\ u \text{ iff } root(t)\ rel_{Node}\ root(q) \\ \qquad \text{and for every transition, } root(u) \stackrel{\alpha}{\rightarrow} u_i, \text{ there exists a transition,} \\ \qquad root(t) \stackrel{\alpha}{\rightarrow} t_j, \text{ such that } t_j\ live_{Tree}\ u_i \end{array}$$

That is, existence of an α-transition in the abstract tree implies that all related concrete trees must be capable of making α-transitions to related nodes.

A liveness analysis for the CCS-notation defined in the previous section requires a different abstraction of a process pool: an abstract pool is a mapping in $Label \rightarrow \{0, 1, \ldots, n\}$, for fixed $n \geq 0$. 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 form of abstract pool is

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

and the abstract semantics rules are the concrete rules along with

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

It is straightforward to prove that the above relation and rule generate abstract trees that are live-related to concrete ones.

The natural language for checking liveness properties is the diamond-mu-calculus:

$$\phi \ ::= \ p \ \mid \ \neg p \ \mid \ \langle\alpha\rangle\phi \ \mid \ \phi_1 \wedge \phi_2 \ \mid \ \phi_1 \vee \phi_2 \ \mid \ \mu x.\phi \ \mid \ \nu x.\phi$$

We have that consistency holds for the diamond-mu-calculus: $t\ live_{Tree}\ u$ and $root(u) \models \phi$ imply $root(t) \models \phi$.

As noted by Dams [12], the set of abstract nodes/pools for the safety and liveness analyses should be the same, as should be the relation, rel_{Node} , because this lets one model check the full mu-calculus, but this appears to be difficult to ensure in practice.

3.2 Argument Transmission via Channels

Now that the basic analysis method is in place, we consider value passing along CCS channels. The syntax is altered to read

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

where $\alpha?x.e$ inputs a value on channel α and binds it to identifier x within scope e, and $\alpha!n.e$ outputs (the value of) numeric expression n on channel α and proceeds with e. The relevant semantics rules are

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

where v is a numeral and τ represents an internal step, a "tau move." To keep the examples small, we will present from here on examples containing only tau moves. But the techniques apply with no restriction to behavior trees with non-tau moves as well.

The above semantics rules define argument transmission via substitution, but substitutions generate new program syntax, which hampers an abstract interpretation. For this reason, we revise the semantics rules into an environment semantics: each process, e, owns a local environment, ρ, that holds bindings of identifiers to numerals, and we write a process configuration $\rho \vdash e$. Further, assume that the evaluation of arithmetic expressions will be performed by an orthogonal set of semantics rules: for environment, ρ, and numeric expression, n, we write $\rho \vdash n \Downarrow v$ to assert that numeric expression n evaluates to numeral value v in environment ρ. 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 8 shows the resulting semantics. To simplify matters, we assume that argument identifiers are distinct from process indentifiers; also, each recursive process, $\mu p(e)$, uses a unique process identifier, p. Environments hold bindings of argument identifiers to numbers and process identifiers to local environments.

The computation rules are kept simple by employing the usual symmetric, monoidal congruences plus three more: the first additional congruence explains how an environment is copied to component processes of a system; the second shows how the semantics rules for numeric-expression evaluation interact with process configurations; and the third describes recursive process unfolding in terms of process-identifier lookup. The computation rules themselves are as

$c \in$ Expression-configuration $\quad \rho \in$ Environment
$e \in$ Expression $\quad n \in$ Arithmetic-expression
$x \in$ Argument-identifier $\quad v \in$ Numeral
$p \in$ Process-identifier $\quad \alpha \in$ Channel

$$\begin{array}{lll} e & ::= & 0 \;\mid\; \alpha?x.e \;\mid\; \alpha!n.e \;\mid\; e_1 \,||\, e_2 \;\mid\; \mu p(e) \;\mid\; p \\ c & ::= & \rho \vdash e \;\mid\; c_1 \,||\, c_2 \\ \rho & ::= & \{x_i = v_i\} \cup \{p_j \mapsto \rho_j\} \end{array}$$

Congruences: usual symmetric, monoidal rules, plus:

$$\rho \vdash (e_1 \,||\, e_2) \equiv (\rho \vdash e_1) \,||\, (\rho \vdash e_2) \qquad \frac{\rho \vdash n \Downarrow v}{\rho \vdash \alpha!n.e \equiv \rho \vdash \alpha!v.e} \qquad \frac{(p \mapsto \rho) \in \rho'}{\rho' \vdash p \equiv \rho \vdash \mu p(e)}$$

Computation rules:

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

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

Figure 8: Semantics for value passing

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 9 displays an example behavior tree.

In order to perform an abstract interpretation, it is helpful to use again the concepts of concrete and abstract process pool. Here, a concrete process pool is a bag of (environment,process) pairs. Assuming that processes are labelled, one can represent a process pool as a function of form $Label \to Bag(Environment)$, where $Environment = (ArgumentId \to Nat) \times (ProcessId \to Environment)$.[13] This suggests that an an abstract process pool is a function of form $Label \to (AbstractEnv \times \{0, 1, \omega\})$, where $AbstractEnv = (ArgumentId \to AbstractNat) \times (ProcessId \to AbstractEnv)$.[14] That is, multiple occurrences of the same process are combined, which means their environments must be joined into one. For example, for the labelling

$$P = \ell_0\!: \mu p\, \ell_1\!: (\ell_2\!: \alpha?x.\ell_3\!: \alpha!x + 1.\ell_4\!: p \,||\, \ell_5\!: p)$$

the configuration $\rho_2 \vdash P \,||\, \rho_3 \vdash \alpha!x + 1.p \,||\, \rho_4 \vdash P$ is encoded as the process pool $[\ell_0 \mapsto \{\rho_2, \rho_4\}][\ell_3 \mapsto \{\rho_3\}](\lambda\ell.\emptyset)$. For the purposes of constant

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

[14] Since the sets $ArgumentId$ and $ProcessId$ are finite, and the language is first order and statically scoped, if $AbstractNat$ has the finite chain property, so must $AbstractEnv$.

Let $P = \mu p(\alpha?x.(\beta!x.0 \mid\mid \alpha!x + 1.0 \mid\mid p))$ and $\rho_i = \{x = i, p \mapsto \emptyset\}$ in

$$\emptyset \vdash (\alpha!3.0 \mid\mid P) \quad \equiv \emptyset \vdash \alpha!3.0 \mid\mid \emptyset \vdash P$$

$$\Big| \; \tau\,(\alpha!3)$$

$$\rho_3 \vdash (\beta!x.0 \mid\mid \alpha!x + 1.0 \mid\mid p) \quad \equiv \rho_3 \vdash \beta!x.0 \mid\mid \rho_3 \vdash \alpha!x + 1.0 \mid\mid \rho_3 \vdash p$$

$$\equiv \rho_3 \vdash \beta!x.0 \mid\mid \rho_3 \vdash \alpha!x + 1.0 \mid\mid \emptyset \vdash P \quad \equiv \rho_3 \vdash \beta!3.0 \mid\mid \rho_3 \vdash \alpha!4.0 \mid\mid \emptyset \vdash P$$

$$\Big| \; \tau\,(\alpha!4)$$

$$\rho_3 \vdash \beta!3.0 \mid\mid \rho_4 \vdash \beta!x.0 \mid\mid \rho_4 \vdash \alpha!x + 1.0 \mid\mid \rho_4 \vdash p$$

$$\equiv \rho_3 \vdash \beta!3.0 \mid\mid \rho_4 \vdash \beta!4.0 \mid\mid \rho_4 \vdash \alpha!5.0 \mid\mid \emptyset \vdash P$$

$$\Big| \; \tau\,(\alpha!5)$$

$$\rho_3 \vdash \beta!3.0 \mid\mid \rho_4 \vdash \beta!4.0 \mid\mid \rho_5 \vdash \beta!x.0 \mid\mid \rho_5 \vdash \alpha!x + 1.0 \mid\mid \rho_5 \vdash p$$

$$\vdots$$

Figure 9: Program with value passing

propagation analysis, the corresponding abstract process pool would appear $[\ell_0 \mapsto (\rho_\top, \omega)][\ell_3 \mapsto (\rho_3, 1)](\lambda\ell.(\bot, 0))$.

The relation between concrete and abstract pools is again a safety relation:

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

where concrete and abstract environments are related in the expected way:[15]

$$\begin{array}{l} (\{x_i = v_i\}, \{p_j \mapsto \rho_j\})\ rel_{Env}\ (\{x_i = a_i\}, \{p_j \mapsto \rho'_j\} \\ \text{iff } (i) \text{ for all } x_i, v_i\ rel_{Nat}\ a_i, \text{ and } (ii) \text{ for all } p_j, \rho_j\ rel_{Env}\ \rho'_j \end{array}$$

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

$$\frac{\rho \vdash e \overset{\Delta}{\to} \rho' \vdash e'}{\rho \vdash e^+ \overset{\Delta}{\to} \rho \vdash e^+ \mid\mid \rho' \vdash e'}$$

Figure 10 shows the constant propagation analysis of the example in the previous figure. Nodes of interest are labelled: node (a) is an unfolding equivalence that can be enacted by the analysis, but node (b) is placed in parentheses because it is for exposition only—the analysis does *not* generate new source program syntax.[16] The transition from (a) must be a τ-step, transmitting $\alpha!4$; The

[15] Although the relation rel_{Env} is defined recusively, the syntax definitions of concrete and abstract environments are defined inductively, so the least relation satisfying the recursive definition can be used.

[16] Nonetheless, an implementation would of course calculate that the x in $\beta!x$ and the $x + 1$ in $\alpha!x + 1$ evaluate to 3 and 4 respectively.

Let $P = \mu p(\alpha?x.(\beta!x.0 \parallel \alpha!x+1.0 \parallel p))$ and $\rho_i = \{x = i, p \mapsto \emptyset\}$ in

$$
\begin{array}{ll}
 & \emptyset \vdash \alpha!3.0 \parallel \emptyset \vdash P \\
 & \qquad\quad \big|\ \tau\ (\alpha!3) \\
 & \rho_3 \vdash \beta!x.0 \parallel \rho_3 \vdash \alpha!x+1.0 \parallel \rho_3 \vdash p \\
\text{(a)} & \equiv \rho_3 \vdash \beta!x.0 \parallel \rho_3 \vdash \alpha!x+1.0 \parallel \emptyset \vdash P \\
\text{(b)} & (\equiv \rho_3 \vdash \beta!3.0 \parallel \rho_3 \vdash \alpha!4.0 \parallel \emptyset \vdash P) \\
 & \qquad\quad \big|\ \tau\ (\alpha!4) \\
\text{(c)} & \rho_\top \vdash (\beta!x.0)^+ \parallel \rho_4 \vdash \alpha!x+1.0 \parallel \rho_4 \vdash p \\
 & \equiv \rho_\top \vdash (\beta!x.0)^+ \parallel \rho_4 \vdash \alpha!x+1.0 \parallel \emptyset \vdash P \\
\text{(d)} & (\equiv \rho_\top \vdash (\beta!\top.0)^+ \parallel \rho_4 \vdash \alpha!5.0 \parallel \emptyset \vdash P) \\
 & \qquad\quad \big|\ \tau\ (\alpha!5) \\
\text{(e)} & \rho_\top \vdash (\beta!x.0)^+ \parallel \rho_5 \vdash \alpha!x+1.0 \parallel \rho_5 \vdash p \\
\text{(f)} & \rho_\top \vdash (\beta!x.0)^+ \parallel \rho_\top \vdash \alpha!x+1.0 \parallel \rho_\top \vdash p \\
 & \equiv \rho_\top \vdash (\beta!x.0)^+ \parallel \rho_\top \vdash \alpha!x+1.0 \parallel \emptyset \vdash P \\
 & (\equiv \rho_\top \vdash (\beta!\top.0)^+ \parallel \rho_\top \vdash \alpha!\top.0 \parallel \emptyset \vdash P) \\
 & \qquad\quad \big|\ \tau\ (\alpha!\top) \\
\text{(g)} & \rho_\top \vdash (\beta!x.0)^+ \parallel \rho_\top \vdash \alpha!x+1.0 \parallel \rho_\top \vdash p
\end{array}
$$

Figure 10: Constant propagation analysis

resulting configuration reads $\rho_3 \vdash \beta!x.0 \parallel \rho_4 \vdash \beta!x.0 \parallel \rho_4 \vdash \alpha!x+1.0 \parallel \rho_4 \vdash p$, but the two occurrences of process $\beta!x.0$ must be combined, producing the abstract process configuration at node (c)—note that $\rho_3 \sqcup \rho_4 = \rho_\top$. Node (d), also parenthesized, states that the x in $\beta!x.0$ evaluates to $\top$. Node (d) is coded as it is because, if $\rho \vdash n \Downarrow v$ holds, then $\rho \vdash (\alpha!n.e)^+$ is bisimular to $\rho \vdash (\alpha!v.e)^+$. Following the transition to node (e), the memoization process notes that the syntax configuration at (e) appeared earlier at (c); this forces the respective environments at nodes (c) and (e) to be joined and produces node (f). Finally, node (f) repeats at node (g), making the tree regular.

The abstract interpretation of the source syntax plus the usual finite chain property of the abstract environment set ensure that the abstract behavior trees must be regular. Because environments hold the numerical values of the argument identifiers, no new source program syntax is generated.

4 Channels as Arguments

It is straightforward to adapt the CCS-variant in the previous section to use channels as arguments; the relevant changes are presented in Figure 11. An

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

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

Congruences: like Figure 8, but replace second congruence by:

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

Computation rules: like Figure 8, but replace first two rules by:

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

Figure 11: Semantics for channels as arguments

example computation is given in Figure 12.

As before, think of a program as a concrete process pool, that is, a function of the form $Label \to Bag(Environment)$, where $Environment = (ArgumentId \to Channel) \times (ProcessId \to Environment)$. A useful abstract interpretation defines an abstract process pool to be a function of the form $Label \to AbstractEnv \times \{0, 1, \omega\}$, where $AbstractEnv = (ArgumentId \to \wp(Channel)) \times (ProcessId \to AbstractEnv)$. That is, in the abstract interpretation, an abstract environment must associate an argument identifier with a set of possible channels that the identifier might denote. Since the set of channels that appear in a given program is finite, $\wp(Channel)$ has the finite chain property.

The relation between concrete and abstract process pools is as before:

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

where concrete and abstract environments are related like before:

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

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

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

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

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

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

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

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

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

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

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

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

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

$\tau(\alpha!\alpha)$

$\vdots$

Figure 12: Program with channel arguments

If we use the congruence rule in Figure 11 to define substitution in the abstract interpretation, 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 computation rules for communication:

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

That is, the entire set of channels, S, is transmitted along any $\alpha \in S$. This is a form of "independent attribute" analysis of channel flow.

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

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

That is, 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, $\{x = \{\alpha,\beta\}\} \vdash x!x.0 \equiv (\{x = \{\alpha,\beta\}\} \vdash \alpha!\alpha.0) + (\{x = \{\alpha,\beta\}\} \vdash \beta!\beta.0)$.[17] 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.

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

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

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

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

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

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

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

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

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

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

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

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

$\tau(\alpha!\alpha)$

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

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

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

Figure 13: Abstract interpretation of channel-argument passing

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

Figure 13 shows the regular tree that results from the relational analysis of the program in Figure 12. 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).

4.1 Channel Creation

In the π-calculus [19], new channels can be created on the fly by using a new construction, $\nu x.e$, where a new channel is created and used for all references to x in e. In this paper, we treat channel creation as an observable action, and we write a substitution semantics for it as follows:

$$\nu x.e \stackrel{\nu\alpha}{\rightarrow} [\alpha/x]e \quad \text{where } \alpha \text{ is globally fresh}$$

$$e \quad ::= \quad \ldots \quad | \quad \nu x.e \quad | \quad (\mu p\ x)(e)$$

Congruences: like Figures 8 and 11, plus:

$$\frac{(y = \alpha,\ p \mapsto \rho) \in \rho'}{\rho' \vdash p(y) \equiv \rho \vdash ((\mu p\ x)(e))(\alpha)}$$

Computation rules: like Figures 8 and 11, but add:

$$\rho \vdash \nu x.e \overset{\nu\alpha}{\to} \rho \oplus \{x = \alpha\} \vdash e, \quad \text{where } \alpha \text{ is globally fresh}$$

$$\frac{\rho \oplus \{x = \alpha\} \vdash e \overset{\Delta}{\to} \rho' \vdash e'}{\rho \vdash (\mu p\ x)(e)(\alpha) \overset{\Delta}{\to} \rho' \oplus \{p \mapsto \rho\} \vdash e'}$$

Figure 14: Channel creation and parameterized recursive processes

By "globally fresh," we mean that α is new to the entire program. (To define this precisely, we should employ a "channel environment" [22], but we use the informal definition here.) To integrate the semantics of channel creation into the environment semantics of Figures 8 and 11, we present the environment-semantics rule for channel creation in Figure 14. For ease in stating the examples in this section, we parameterize recursive processes, $\mu p(e)$, on a single channel argument, x, giving $(\mu p\ x)(e)$. The congruences and computation rules for parameterized processes are presented in Figure 14 as well. Figure 15 gives an example computation. The example computation shows how multiple new channels can be created in the course of the computation. Say that the *origin* of a channel is the νx that created it. (Global channels have themselves as their "origins.") Since we work with labelled expressions, we use the label, ℓ, of $\ell\colon \nu x.e$ to denote the origin of the channel(s) created by the νx.

Our abstract interpretation of channel creation cannot track precisely the number of channels created, but it can trace the paths of communications formed by those channels created by the νx constructions. When the labelled expression, $\ell\colon \nu x.e$ is abstractly interpreted, the origin, ℓ, is used to represent the created channel. Of course, this means that subsequent executions of the same $\nu x.e$ generate a new channel named by the same origin, ℓ, causing a loss of precision of the number of channels created and the control paths within the program.[18] But this simplistic analysis has a simple and natural correctness criterion:

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

[18] Alternatively, one can name the newly created channel with the computation path in the behavior tree which led to the channel's creation [31].

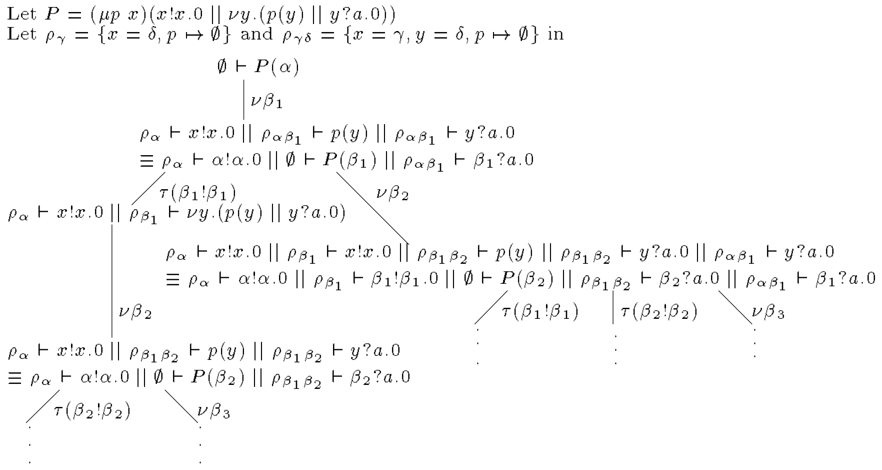

Figure 15: Computation with channel creation

where concrete and abstract environments are related as follows:

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

Given our simplistic abstract interpretation of syntax, where a process can occur 0, 1, or ω times, the above modelling suffices.

Figure 16 gives the abstract interpretation of the example in Figure 15, where ℓ is used as the label of the ν-expression in the example.

5 Related Work

The direct antecedent of the syntax abstraction technique in this paper is the "star abstraction" technique of Codish, Falaschi, and Marriott [5, 6], where the size of an unfolded logic program is controlled by using what is essentially the "+" operator shown in the examples of this paper. It is important to note that the regular-expression-like abstraction of syntax in this paper was chosen primarily for ease of exposition. A natural extension would use a context-free grammar to abstract a program's syntax.[19] Indeed, a similar approach was undertaken by Giannotti and Latella [16], where the concrete semantics of an equationally-defined program is taken to be the Herbrand universe of its terms, and the abstract semantics is a finite subset of "partial terms," e.g., the program $add(0, y) = y \mid add(s(x), y) = s(add(x, y)$ is abstractly interpreted over the set of partial terms $\{0, s(0), \top, s(\top)\}$, where the partial terms were extracted from the definition of add.

[19] This suggestion is due to Patrick Cousot.

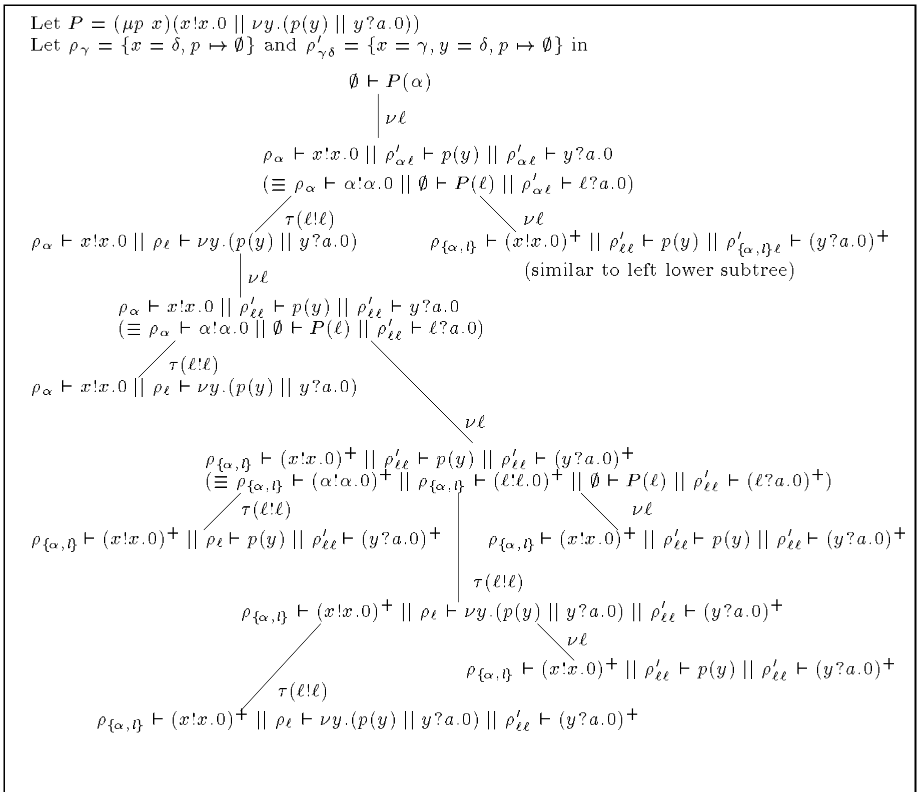

Figure 16: Abstract interpretation of channel creation

The papers of Colby [7] and Dwyer [15] on analysis of concurrent programs are also relevant; in particular, Colby applies both abstraction and partitioning techniques on program trees.

The primary application of the techniques in this paper is calculation of the abstract collecting semantics of a program; such information is essential for code improvement. But collecting-semantics-based code improvement is subsumed by model checking [28, 29], so we briefly review relevant research in applying abstract interpretation to model checking and in model checking processes with infinite state spaces.

The canonical work in abstract interpretation of models for model checking is that of Clarke, Grumberg, and Long [3]. Starting with a (possibly infinite) set of concrete states, C, and a concrete state transition relation, $R_C \subseteq C \times C$, one proposes a finite set of abstract states, A, and a surjective map, $\beta : C \rightarrow A$; one obtains the abstract transition relation, $R_A(a, a')$ iff there exist $c, c' \in C$ such that $\beta(c) = a, \beta(c') = a'$, and $R_C(c, c')$. Thus, R_A is a simulation of R_C, and properties in $\forall$CTL* can be checked on the abstract model. It is crucial that the abstract state set is finite, because no memoization techniques are used to force regular tree creation.

This basic approach is examined by Bensalem, Bouajjani, Loiseaux, and Sifakis [1] and Dams, Grumberg, and Gerth [13], where Galois connection machinery is used in place of the β map.

All these lines of work note that the dual of the usual simulation gives a technique for liveness property checking with ∃CTL*, and a bisimulation allows one to check full CTL*. Dams [12] notes that a bisimulation is not essential for checking CTL*: one needs only simulations in both directions where the same abstract state set is used for both simulations. Cleaveland, Iyer, and Yankelevich [4] formalize this idea.

Models with infinite state spaces are handled directly by Hungar and Steffen [17], where a process is a context free grammar, or equivalently, a finite state automaton with "subroutine calls" to other automata. Such processes define the context free languages. Both iterative [2] and tableau-based [17] model-checking algorithms are given for this format; the algorithms are complete for the alternation-free mu-calculus. Note that Hungar and Steffen are *not* concerned with discovering a context-free representation (if any) for an arbitrary CCS-specification.

Another representative direction of research is that of De Francesco, Fantechi, Gnesi, and Inverardi [14], where arbitrary CCS-specifications (including those that define non-context-free languages) with infinite state spaces are handled directly with least-fixed point theory: for a CCS program, p, its infinite state transition system, $SS(p)$, is the limit of its finite approximations (unfoldings), $SS_i(p)$, $i \geq 0$. One can do model checking directly upon the $SS_i(p)$, for $i = 0, 1, \ldots$, and verify a useful subset of (a variant of) CTL*. For practical use, it is crucial that repetition nodes are located as quickly as possible in the generation of the $SS_i(p)$s.

5.1 Conclusions, Applications, and Future Work

As stated in the Introduction, the primary goal of this paper was to extend simply and elegantly abstract interpretation techniques to modern programming languages defined by structured operational semantics. The net result is a clear methodology for generating semantically safe (and live) abstract, regular trees for programs that do not possess obvious state-transition diagram depictions. This widens the application of the work of Steffen [28], for example. A new insight is that one can employ memoization on abstract interpretations with infinite data sets to generate finite structures for model checking; this has not been attempted in earlier research. Also, the work here makes clear that the abstraction of program syntax can play a central role in classical abstract interpretation.

The primary application of this research is to the validation of safety properties of programs, whether these properties are validated by collecting-semantics techniques or by model checking. An implementation of techniques similar to the ones described in this paper is already underway by Cridlig [11].

It is a greater challenge to formulate useful liveness checking techniques for modern languages defined with structured operational semantics: Since liveness is the abstract interpretive dual of safety, semantic joins of abstract data at control-flow join points (e.g., the arms of a conditional) must be replaced by semantic meets. This requires a rethinking of the classical abstract data sets used with data-flow analyses and is certainly deserving of future work.

It is also planned to apply the techniques in this paper to model checking linkage specifications of statically typed, modular systems.

6 Acknowledgements

I must first thank Ed Clarke for his interest in this work and his comments on related research. Patrick Cousot, Mads Dam, Rocco De Nicola, and Giorgio Levi also provided helpful comments and pointers to the literature. The anonymous referee is thanked for his insightful commentary.

References

[1] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In G. vanBochman and D. Probst, editors, *Computer Aided Verification: CAV'92*, number 663 in Lecture Notes in Computer Science, pages 260–273. Springer-Verlag, 1992.

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

[3] 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.

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

[5] M. Codish, S. Debray, and R. Giacobazzi. Compositional analysis of modular logic programs. In *Proc. 20th ACM Symp. on Principles of Programming Languages*, pages 451–464. ACM Press, 1993.

[6] 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.

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

[8] 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.

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

[10] 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.

[11] R. Cridlig. Implementing a static analyzer of concurrent programs: problems and perspectives. In M. Dam and F. Orava, editors, *Proc. LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, Lecture Notes in Computer Science. Springer, 1996.

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

[13] 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.

[14] N. DeFrancesco, A. Fantechi, S. Gnesi, and P. Inverardi. Model checking of non-finite state processes by finite approximations. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, Lecture Notes in Computer Science 1019, pages 195–215. Springer, 1995.

[15] M. Dwyer and L. Clark. Data flow analysis for verifying properties of concurrent programs. In *Proc. 2d ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 62–75. ACM Press, 1994.

[16] 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.

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

[18] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 2–117. Oxford University Press, 1992.

[19] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Lab. for Foundations of Computer Science, University of Edinburgh, 1991. Also in Logic and Algebra of Specification, F.L.Bauer, ed., Springer, 1993.

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

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

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

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

[24] 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.

[25] 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.

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

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

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

[29] 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.

[30] 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.

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