Analyzing the Communication Topology of Concurrent Programs

Christopher Colby*
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213-3891
colby+@cmu.edu

Abstract

Concurrent languages present complex problems for program analysis. Existing analyses are either imprecise, exponential, or apply only to languages with statically-allocated processes and channels. We present a new polynomial-time analysis using abstract interpretation that addresses the general problem of determining the communication topology of programs in a subset of Concurrent ML with arbitrary data structures, recursive higher-order functions, dynamic processor allocation, dynamic channel creation, and synchronous message-passing operations `transmit` and `receive`. The analysis addresses the following question: *Which occurrences of* `transmit` *can match which occurrences of* `receive`*?* The notion of occurrence is formalized as a control path in a small-step semantics, which provides a powerful basis for distinguishing recursive communication topologies. The analysis is *relational*, in that it relates pairs of processes, and *non-uniform*, in that it distinguishes between iterations in an infinite recursive communication pattern. The results are thus precise enough to aid processor allocation, scheduling, and sequentialization on both uniprocessors and multiprocessors.

1 Introduction

Analysis of concurrent languages has proven to be a challenging task. The vast majority of existing work (*e.g.*, [CC80, Mer91]) deals with languages that do not have dynamic processor allocation and dynamic channel creation. However, we feel that these are important features of a realistic concurrent language. Unfortunately, they present difficult problems for analyses; consequently, little work has been done in this area.

We present a new polynomial-time analysis, using abstract interpretation [CC77], that addresses the general problem of determining the communication topology of programs in a powerful subset of Concurrent ML.

Concurrent ML [Rep92] is an extension of Standard ML [MTH90] that provides dynamically-created *processes* and dynamically-created first-class *channels*. The fundamental restrictions on our subset are that there are no references (mutable values) or exceptions, and that the only two message-passing operations are `transmit` and `receive`. Note that references may be modeled by message-passing [Rep92].

Our analysis addresses the following question: *Which occurrences of* `transmit` *can match which occurrences of* `receive`*?* To illustrate, consider the following partially-labeled program that recursively and concurrently folds the operations `base` and `combine` over a binary tree and transmits the result through an output channel.

```
datatype 'a Tree = LEAF of 'a
                 | NODE of ('a Tree * 'a Tree)

rec fold (ch,tree) =
  case tree of
    LEAF v => [T]:transmit (ch, base v)
  | NODE (ltree,rtree) =>
      let val l = [L]:channel ()
          val r = [R]:channel ()
      in [SL]:spawn (fold (l, ltree));
         [SR]:spawn (fold (r, rtree));
         [T']:transmit (ch, combine ([RL]:receive l,
                                     [RR]:receive r))
      end
```

The analysis determines that any dynamic occurrence of the receive at $\boxed{\mathrm{RL}}$ can be matched *only* with the transmit at $\boxed{\mathrm{T}}$ or $\boxed{\mathrm{T'}}$ in the process created at the *dynamically preceding* spawn at $\boxed{\mathrm{SL}}$. Likewise for $\boxed{\mathrm{RR}}$ and $\boxed{\mathrm{SR}}$, and thus the program exhibits a tree topology of communication. The analysis determines this information regardless of the remainder of the program; in particular, the functions `base` and `combine` may spawn processes, create and use new channels, and even call `fold`.

Even for sequential languages, polyvariance (*e.g.*, [JM82]) is insufficient to distinguish between iterations in an infinite pattern. For this, we need *non-uniform* analysis techniques (*e.g.*, [Deu92, Chp. 4]). In the context of communication topology, we identify an analysis as non-uniform if it can distinguish between iterations in a recursive communication pattern. In this case, the pattern is an infinite tree in which messages are sent only from a child to its parent. This is very precise information and can be exploited in the following ways in a compiler:

- Process allocation on a multiprocessor. Messages are sent only from a child to its parent, so an appropriate

*Visiting the Laboratoire d'Informatique, École Polytechnique, 91128 Palaiseau cedex, France allocation strategy to minimize the cost of this virtual one-way tree topology may be chosen.

- Sequentialization on both uniprocessors and multiprocessors. The topology information given above is *necessary* to inline the two spawns in this example, and a weaker analysis would be insufficient. By switching to sequentialized code when the physical processors are exhausted, the overhead of virtual processes can be eliminated. This overhead can be especially costly on multiprocessors that must access shared resources for virtual-processor management.

The details of such optimizations are non-trivial and beyond the scope of this paper. Our aim is rather to provide a clear description of the analysis itself.

2 Related Work

In [CH92], Chow and Harrison present an analysis for a shared-memory language with a **cobegin-coend** construct that synchronizes on completion of its branches. This offers only a limited form of dynamic processor allocation. Their analysis is exponential, and because it is based on a uniform location model, it cannot determine recursive topologies. In [JW94], Jagannathan and Weeks present an analysis for a language with dynamic processor allocation and asynchronous communication via shared memory. Although the analysis is polynomial, it is by in large an extension of a uniform control-flow analysis for higher-order sequential languages [JM82]; as such, the results give little information about the concurrency present in the program. In [NN94], Nielson and Nielson present an analysis based on a type system for a large subset of Concurrent ML to address the question of whether or not a program will generate an infinite number of channels or processes. The analysis gives no further information about the communication topology of programs.

To our knowledge, our work is the first non-uniform analysis for a language with dynamic processor allocation.

3 A Sequential Core Language

We begin by defining a sequential core language. It is the fragment of Standard ML without references described below, with datatypes, records, and higher-order recursive functions.

$$k \in \mathit{Const} \quad x, y \in \mathit{Var} \quad f \in \mathit{Field} \quad c \in \mathit{Constructor}$$

$$\begin{array}{rcl} P, e & ::= & k \mid x \mid \{f_1\texttt{=}e_1\texttt{,} \ldots \texttt{,}f_n\texttt{=}e_n\} \mid op\ e \\ & & \texttt{fn}\ x\ \texttt{=>}\ e \mid \texttt{rec}\ y(x)\ \texttt{=}\ e \mid e\ e' \mid \\ & & \texttt{case}\ e\ \texttt{of}\ c_1\, x_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\, x_n\ \texttt{=>}\ e_n \\ op & ::= & c \mid \#f \end{array}$$

A tuple $(e_1, \ldots, e_n)$ is syntactic sugar for $\{1\texttt{=}e_1, \ldots, n\texttt{=}e_n\}$, and $\{\}$ is the syntax for the unit value. The only constructs of Standard ML omitted here for technical reasons are references and exceptions. All other omissions, such as other primitive operations and type information, are for reasons of brevity and clarity.

There are many techniques for designing a dynamic semantics of a sequential functional language. In Section 4, however, we will be extending this sequential core with concurrency constructs, and it is difficult to model concurrency with denotational semantics or structured operational semantics. We therefore choose a small-step operational semantics for the sequential core. In some approaches to operational semantics, transitions are defined between terms in the language. We instead choose an abstract-machine semantics in which transitions are defined between states comprising (1) a *program address* representing the current syntactic point of evaluation (sometimes called the program point or label) and (2) a *frame* representing the other necessary state information.

Each syntactic occurrence of an expression in program P has a unique program address corresponding to its position in the syntax tree of P. We write $P@a$ to denote the expression at program address a. The form of program addresses follows the structure of the syntax, where the program address of P itself is $\bullet$ (*i.e.*, $P@\bullet = P$).

If $P@a = \{f_1\texttt{=}e_1, \ldots, f_n\texttt{=}e_n\}$
  then $P@a.\text{field}_i = e_i$.
If $P@a = \texttt{fn}\ x\ \texttt{=>}\ e$ or $P@a = \texttt{rec}\ y(x)\ \texttt{=}\ e$
  then $P@a.\text{body} = e$.
If $P@a = e\ e'$
  then $P@a.\text{op} = e$ and $P@a.\text{arg} = e'$.
If $P@a = op\ e$
  then $P@a.\text{arg} = e$.
If $P@a = \texttt{case}\ e\ \texttt{of}\ c_1\, x_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\, x_n\ \texttt{=>}\ e_n$
  then $P@a.\text{pred} = e$ and $P@a.\text{case}_i = e_i$.

Later, we will define *control paths* as sequences of program addresses; control paths will play a crucial role for the concurrent language semantics in Section 4 and for the approximation in Section 5 and are thus the chief motivation for introducing program addresses at this stage.

For each program address a there is a unique variable $\langle a\rangle$ used to represent the value of expression $P@a$. Note that this is static; if there are m function-bound and n expressions in P, then including these program-address variables there are $n+m$ variables total.

A state $a\!:\!(\rho,\kappa)$ has a program address a, representing the current syntactic point of evaluation, and a frame (ρ,κ) comprising an environment ρ, which is a partial map assigning values to variables, and a continuation κ, which is either $\bullet$ at top level or an address to return to and a frame to restore upon return of a function call. Values are constants, records, datatypes, and closures. A closure (a,ρ) is a pair of the program address of its function and an environment assigning values to its free variables.

There is a transition for each possible single step of a strict left-to-right evaluation of P. The semantic domains and transitions are defined in Figure 1. Given a state $a\!:\!(\rho,\kappa)$, we sometimes write $\overline{a}\!:\!(\rho,\kappa)$ to assert that $\rho(\langle a\rangle)$ is undefined and $\underline{a}\!:\!(\rho,\kappa)$ to assert that $\rho(\langle a\rangle)$ is defined. The former corresponds to a state at the beginning of the evaluation of expression $P@a$, and the latter corresponds to a state at the end of the evaluation of expression $P@a$.

A point about this semantics is that "old" bindings are never removed from the environment. For instance, if the body of the program is $((e_1, e_2), e_3)$, then during the evaluation of e_3 the environment will still contain bindings for the program addresses of e_1 and e_2. But these bindings are no longer needed; only the binding for the program address of (e_1, e_2) is needed. This choice simplifies the formulation without affecting the results of the analysis. The alternative would be to explicitly remove unneeded bindings in the appropriate transitions (*e.g.*, for the case above, the transition

Semantic Sets:

$$\begin{array}{rllll}
\text{program addresses} & (\mathit{Addr}): & a & ::= & \bullet \mid a.\text{field}_i \mid a.\text{body} \mid a.\text{op} \mid a.\text{arg} \mid a.\text{pred} \mid a.\text{case}_i \\
\text{values} & (\mathit{Val}): & v & ::= & k \mid \{f_1 = v_1, \ldots, f_n = v_n\} \mid c\ v \mid (a, \rho) \\
\text{environments} & (\mathit{Env} = \mathit{Var} \rightharpoonup \mathit{Val}): & \rho & ::= & \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \\
\text{continuations} & (\mathit{Kont} = 1 + \mathit{State}): & \kappa & ::= & \bullet \mid s \\
\text{frames} & (\mathit{Frame} = \mathit{Env} \times \mathit{Kont}): & \pi & ::= & (\rho, \kappa) \\
\text{states} & (\mathit{State} = \mathit{Addr} \times \mathit{Frame}): & s & ::= & a : \pi \\
\text{transitions:} & & & & s \longmapsto s'
\end{array}$$

Notation: $\{\ldots, f = v, \ldots\}.f = v$

Initial State: $s_{init} = \bullet : (\{\}, \bullet)$

Transitions:

$$\begin{array}{rcll}
\overline{a} : (\rho, \kappa) & \longmapsto & a : (\rho[\langle a \rangle \mapsto k], \kappa) & \text{if } P@a = k \\
\\
\overline{a} : (\rho, \kappa) & \longmapsto & a : (\rho[\langle a \rangle \mapsto \rho(x)], \kappa) & \text{if } P@a = x \\
\\
\overline{a} : (\rho, \kappa) & \longmapsto & a : (\rho[\langle a \rangle \mapsto \{\}], \kappa) & \text{if } P@a = \{\} \\
\overline{a} : (\rho, \kappa) & \longmapsto & a.\text{field}_1 : (\rho, \kappa) & \text{if } P@a = \{f_1\texttt{=}e_1, \ldots, f_n\texttt{=}e_n\} \\
\underline{a.\text{field}_i} : (\rho, \kappa) & \longmapsto & a.\text{field}_{i+1} : (\rho, \kappa) & \text{if } P@a = \{f_1\texttt{=}e_1, \ldots, f_n\texttt{=}e_n\} \ \wedge\ 1 \le i < n \\
\underline{a.\text{field}_n} : (\rho, \kappa) & \longmapsto & a : (\rho[\langle a \rangle \mapsto \{f_1 = \rho(\langle a.\text{field}_1 \rangle), \ldots, f_n = \rho(\langle a.\text{field}_n \rangle)\}], \kappa) & \\
& & & \text{if } P@a = \{f_1\texttt{=}e_1, \ldots, f_n\texttt{=}e_n\} \\
\\
\overline{a} : (\rho, \kappa) & \longmapsto & a.\text{pred} : (\rho, \kappa) & \text{if } P@a = \texttt{case}\ldots \\
a.\text{pred} : (\rho, \kappa) & \longmapsto & a.\text{case}_i : (\rho[x_i \mapsto v], \kappa) & \text{if } P@a = \texttt{case}\ e\ \texttt{of}\ \ldots \mid c_i\ x_i \texttt{=>} e_i \mid \ldots \\
& & & \quad \wedge\ \rho(\langle a.\text{pred} \rangle) = c_i\ v \\
\\
\underline{a.\text{case}_i} : (\rho, \kappa) & \longmapsto & a : (\rho[\langle a \rangle \mapsto \rho(\langle a.\text{case}_i \rangle)], \kappa) & \\
\\
\overline{a} : (\rho, \kappa) & \longmapsto & a.\text{arg} : (\rho, \kappa) & \text{if } P@a = \mathit{op}\ e \\
\underline{a.\text{arg}} : (\rho, \kappa) & \longmapsto & a : (\rho[\langle a \rangle \mapsto c\ (\rho(\langle a.\text{arg} \rangle))], \kappa) & \text{if } P@a = c\ e \\
\underline{a.\text{arg}} : (\rho, \kappa) & \longmapsto & a : (\rho[\langle a \rangle \mapsto \rho(\langle a.\text{arg} \rangle).f], \kappa) & \text{if } P@a = \texttt{\#}f\ e \\
\\
\overline{a} : (\rho, \kappa) & \longmapsto & a : (\rho[\langle a \rangle \mapsto (a, \rho)], \kappa) & \text{if } P@a = \texttt{fn}\ x\ \texttt{=>}\ e\ \vee\ P@a = \texttt{rec}\ y(x)\ \texttt{=}\ e \\
\\
\overline{a} : (\rho, \kappa) & \longmapsto & a.\text{op} : (\rho, \kappa) & \text{if } P@a = e\ e' \\
\underline{a.\text{op}} : (\rho, \kappa) & \longmapsto & a.\text{arg} : (\rho, \kappa) & \\
\underline{a.\text{arg}} : (\rho, \kappa) & \longmapsto & a'.\text{body} : (\rho'[x \mapsto \rho(\langle a.\text{arg} \rangle)], a : (\rho, \kappa)) & \text{if } P@a = e\ e' \ \wedge\ P@a' = \texttt{fn}\ x\ \texttt{=>}\ e'' \\
& & & \quad \wedge\ \rho(\langle a.\text{op} \rangle) = (a', \rho') \\
\underline{a.\text{arg}} : (\rho, \kappa) & \longmapsto & a'.\text{body} : (\rho'[x \mapsto \rho(\langle a.\text{arg} \rangle)][y \mapsto (a', \rho')], a : (\rho, \kappa)) & \text{if } P@a = e\ e' \ \wedge\ P@a' = \texttt{rec}\ y(x)\ \texttt{=}\ e'' \\
& & & \quad \wedge\ \rho(\langle a.\text{op} \rangle) = (a', \rho') \\
\underline{a.\text{body}} : (\rho, a' : (\rho', \kappa)) & \longmapsto & a' : (\rho'[\langle a' \rangle \mapsto \rho(\langle a.\text{body} \rangle)], \kappa) &
\end{array}$$

Figure 1: The dynamic semantics of the sequential core language.

with left-hand-side $a.\text{field}_n : (\rho, \kappa)$).

Example 1 *Suppose the expression* `(`x`,45)` *is at program address a in P. (Recall that this is syntactic sugar for* `{1=`x`,2=45}`*.) The following is the sequence of states in the evaluation of this expression from a state with an environment mapping x to v and with continuation κ. We let $a_1 = a.\text{field}_1$ and $a_2 = a.\text{field}_2$. The environment in the final state maps $\langle a \rangle$ to the result.*

$$\begin{array}{rl}
& a : (\{x \mapsto v\}, \kappa) \\
\longmapsto & a_1 : (\{x \mapsto v\}, \kappa) \\
\longmapsto & a_1 : (\{x \mapsto v, \langle a_1 \rangle \mapsto v\}, \kappa) \\
\longmapsto & a_2 : (\{x \mapsto v, \langle a_1 \rangle \mapsto v\}, \kappa) \\
\longmapsto & a_2 : (\{x \mapsto v, \langle a_1 \rangle \mapsto v, \langle a_2 \rangle \mapsto 45\}, \kappa) \\
\longmapsto & a : (\{x \mapsto v, \langle a_1 \rangle \mapsto v, \langle a_2 \rangle \mapsto 45, \\
& \quad \langle a \rangle \mapsto \{1 = v, 2 = 45\}\}, \kappa)
\end{array}$$

4 Adding Concurrency Constructs

We extend our language with constructs for process generation and synchronous message passing.

$$\begin{array}{rcl}
e & ::= & \ldots \mid \texttt{spawn}\ e \\
op & ::= & \ldots \mid \texttt{channel} \mid \texttt{transmit} \mid \texttt{receive}
\end{array}$$

The expression `spawn` e does not evaluate e directly, but creates a new process to concurrently evaluate e and returns {} to the current process. The primitive operation `channel` takes the unit value and returns a fresh channel (*i.e.*, one that is distinct from any other channel in the current state of evaluation). Values are (synchronously) passed between processes through channels via the primitive operations `transmit` and `receive`. The primitive operation `transmit` takes a pair (v_1, v_2), where v_1 is a channel and v_2

is the value to be transmitted, and blocks until it can synchronize with a matching receiver. The primitive operation `receive` takes a channel and blocks until it can synchronize with a matching transmitter. A blocked transmitter and blocked receiver match when they share the same channel, and when this occurs they may become unblocked and the value atomically transmitted. Note that a blocked transmitter may match with several blocked receivers, and *vice versa*; in these cases, any match is a valid transition.

4.1 Control paths

A *control path* is a sequence $a_1 \ldots a_n$ of program addresses where $a_1 = \bullet$, the program address of the initial state, and for all $1 \leq i < n$ there is a transition from program address a_i to program address a_{i+1}. At any state in the evaluation, each process has a control path: the sequence of program addresses that starts with $\bullet$, traverses the history of evaluation of the process (including its ancestors from oldest to youngest), and ends with the program point at the current state of the process. Control paths serve two purposes: (1) they replace the usual arbitrary set of process identifiers, and (2) they provide a way of modeling channels that supports reasoning about communication topology.

Most formulations of the semantics of concurrent languages (*e.g.*, [BMT92, Rep92]) use an arbitrary set of process identifiers to distinguish processes. In those semantics, when a new process is created an arbitrary fresh process identifier is chosen and attached to that process. We instead make the observation that in any state the current control paths of the processes are guaranteed to be distinct. So instead of attaching a process identifier to a process, we retain its control path[1].

Our motivation for using control paths to replace the usual arbitrary set of process identifiers stems from the observation that process identifiers by themselves have no semantic meaning, while control paths provide a way of modeling channels that provides a powerful basis for reasoning about their *equality*: the key property needed to determine the communication topology of the program. The idea is simple; a channel is modeled as the control path at which it was created. Equality of channels thus reduces to equality of control paths, and control paths provide the rich structure needed for useful abstraction.

4.2 The dynamic semantics

The dynamic semantics of the full concurrent language is shown in Figure 2. A state in the concurrent semantics is a *tree* of program addresses with a frame attached to each leaf, one for each process. The root of the tree is the initial program address $\bullet$, and the path from the root to a leaf is the control path of the process at that leaf. A branch in the tree corresponds to a process spawn. The syntax of program addresses is extended for the spawn expression; if $P@a = \mathtt{spawn}\ e$ then the address of e is a.proc.

The set of values is extended with channel values, which are control paths.

The first rule of the semantics handles the sequential core of the language by extending each sequential transition to its concurrent form. It says that *any* process ready to make a transition from the sequential core may do so, and its control path is extended by one program address.

[1]Indeed, process identifiers can be viewed as abstractions of the control paths.

The second rule handles channel creation. The new channel is the control path at which it is created.

The third rule handles process spawns. A process ready to spawn may create a new branch in the state. The left-hand side of the branch is the first state of the new process; the branch for the new process does not require the continuation, so it is discarded, but it inherits the environment. The control path of the new process is not simply a.proc, but rather $\mathbf{a}a(a.\text{proc})$; this is because a control path of a process starts at the beginning of the evaluation rather than at the beginning of the new process. The existing process continues on the right-hand side of the branch with control path $\mathbf{a}aa$.

The fourth and final rule handles synchronization. The left-hand side of the transition shows two different processes that originally branched from a spawn at control path $\mathbf{a}a_s$, one of which is waiting to perform a transmit, the other of which is waiting to perform a receive. If their channels are equal, then the transition may be taken, in which case a value is transmitted between processes and both take a step representing the returns of their respective operations. The `transmit` expression returns {} and the `receive` expression returns the transmitted value. Recall that the argument of the transmit operation is a pair of the channel and the transmitted value; hence the record-field selectors 1 and 2 in the rule.

5 A Finite Approximation of a Set of States

We wish to compute properties of states. A property is simply the set of states for which it holds, and so the first step in the development of an algorithm to compute these properties is the design of a finite representation of sets of states. Of course, the representation that we choose will exactly determine the properties that it can express, and so this is the most important step of the algorithm design.

Figure 3 shows the definition of an approximation of a set of states with a member of

$$\widehat{State} = SAddr \rightarrow (Branch \times Addr) \rightarrow \widehat{Frame},$$

where $SAddr$ is the set of program addresses of spawn expressions, $Branch = \{\lhd, \rhd\} \times \widehat{CP}$, and $\widehat{Frame}$ is an approximation of sets of frames. Here, $\widehat{CP}$ is a partitioning of the set CP of control paths with $partition \in CP \rightarrow \widehat{CP}$ and is a parameter of the approximation.

A member X of this set can be seen as a four-dimensional table. Say that a set S of states is approximated by $X \in \widehat{State}$. Then given

- a program address a_s such that $P@a_s = \mathtt{spawn}\ e$,
- a token z chosen from $\{\lhd, \rhd\}$,
- a partition $\hat{\mathbf{a}}$ of control paths taken from $\widehat{CP}$, and
- a program address a,

the value $X(a_s)((z, \hat{\mathbf{a}}), a)$ is an abstract frame $\hat{\pi} \in \widehat{Frame}$ that approximates the set of all frames π such that

- if $z = \lhd$ then $\lhd\,\mathbf{a}\dashv a_s \left(\lhd\,\mathbf{a}' \dashv a : \pi \;\middle\|\; s \right) \in S$ for some $\mathbf{a}$, $\mathbf{a}'$, and s such that $partition(\mathbf{a}'a) = \hat{\mathbf{a}}$, and

Semantic Sets:

$$\begin{array}{rlrcl}
\text{program addresses} & (Addr): & a & ::= & \ldots \mid a.\text{proc} \\
\text{values} & (Val): & v & ::= & \ldots \mid \mathbf{a} \\
\text{control paths} & (CP = Addr^*): & \mathbf{a} & ::= & \epsilon \mid \mathbf{a}a \\
\text{states} & (State): & s & ::= & \mathbf{a}:\pi \mid \mathbf{a}\left(s_1 \,\middle\|\, s_2\right) \\
\text{transitions:} & & & & s \Longrightarrow s'
\end{array}$$

Notation: The picture $\lhd\,\mathbf{a}\vdash s$ is a metavariable for a state with subtree s at control path $\mathbf{a}$. We define the family of these sets inductively:

$$\lhd\,\mathbf{a}\mathbf{a}'\vdash s \;::=\; \mathbf{a}\mathbf{a}'s \mid \mathbf{a}\left(\lhd\,\mathbf{a}'\vdash s \,\middle\|\, s'\right) \mid \mathbf{a}\left(s' \,\middle\|\, \lhd\,\mathbf{a}'\vdash s\right)$$

Initial State: (same as the sequential semantics)

Transitions:

$$\lhd\,\mathbf{a}\vdash a:\pi \;\Longrightarrow\; \lhd\,\mathbf{a}\vdash aa':\pi' \qquad \text{if } a:\pi \longmapsto a':\pi'$$

$$\lhd\,\mathbf{a}\vdash \overline{a}:(\rho,\kappa) \;\Longrightarrow\; \lhd\,\mathbf{a}\vdash aa:(\rho[\langle a\rangle \mapsto \mathbf{a}a],\kappa) \qquad \text{if } P@a = \texttt{channel}\ e$$

$$\lhd\,\mathbf{a}\vdash \overline{a}:(\rho,\kappa) \;\Longrightarrow\; \lhd\,\mathbf{a}\vdash a\left(a.\text{proc}:(\rho,\bullet) \,\middle\|\, a:(\rho[\langle a\rangle \mapsto \{\}],\kappa)\right) \qquad \text{if } P@a = \texttt{spawn}\ e$$

$$\lhd\,\mathbf{a}\vdash a_s\left(s_\lhd \,\middle\|\, s_\rhd\right) \;\Longrightarrow\; \lhd\,\mathbf{a}\vdash a_s\left(s'_\lhd \,\middle\|\, s'_\rhd\right) \qquad \begin{array}{l}\text{if } P@a_t = \texttt{transmit}\ e \\ \wedge\ P@a_r = \texttt{receive}\ e' \\ \wedge\ \rho_t(\langle a_t.\text{arg}\rangle).1 = \rho_r(\langle a_r.\text{arg}\rangle) \\ \wedge\ (z_t, z_r) \in \{(\lhd,\rhd),(\rhd,\lhd)\}\end{array}$$

$$s_{z_t} = \lhd\,\mathbf{a}_t\vdash a_t.\text{arg}:(\rho_t,\kappa_t) \qquad s'_{z_t} = \lhd\,\mathbf{a}_t\vdash (a_t.\text{arg})a_t:(\rho_t[\langle a_t\rangle \mapsto \{\}],\kappa_t)$$

$$s_{z_r} = \lhd\,\mathbf{a}_r\vdash a_r.\text{arg}:(\rho_r,\kappa_r) \qquad s'_{z_r} = \lhd\,\mathbf{a}_r\vdash (a_r.\text{arg})a_r:(\rho_r[\langle a_r\rangle \mapsto \rho_t(\langle a_t.\text{arg}\rangle).2],\kappa_r)$$

Figure 2: The dynamic semantics of the full concurrent language.

- if $z = \rhd$ then $\lhd\,\mathbf{a}\vdash a_s\left(s \,\middle\|\, \lhd\,\mathbf{a}'\vdash a:\pi\right) \in S$ for some $\mathbf{a}$, $\mathbf{a}'$, and s such that $partition(\mathbf{a}'a) = \hat{\mathbf{a}}$.

Intuitively, $\hat{\pi}$ approximates the set of all frames that appear in some process in some state in S that is currently at program address a, whose control path went through either the left side of a spawn branch at program address a_s if $z = \lhd$ (new process, beginning with a_s.proc) or the right side if $z = \rhd$ (existing process, beginning with a_s), and whose control-path suffix since that point is in partition $\hat{\mathbf{a}} \in \widehat{CP}$.

This representation was the inspiration for introducing control paths in the semantics in the first place, and particularly for modeling a channel as the control path of it creation point. Our representation is designed to allow the differentiation between channels (and in general any data) created during the evaluation of the left side of a branch, created during the evaluation of the right side of a branch, and created at other points. This differentiation is achieved with the last step of the abstraction, defined by γ_3 and described in Section 5.2. But first, we present the first two stages of the abstraction, outlined at the top of Figure 3.

5.1 Relating processes with their branching points

The function α_1 takes a set of states and extracts from each state all processes in that state, split at the branching points in their control paths. Formally, $(\mathbf{a}, \lhd, \mathbf{a}', \pi) \in \alpha_1(S)$ iff there is some state

$$\lhd\,\mathbf{a}\vdash \epsilon\left(\lhd\,\mathbf{a}'\vdash \epsilon:\pi \,\middle\|\, s\right) \in S,$$

and $(\mathbf{a}, \rhd, \mathbf{a}', \pi) \in \alpha_1(S)$ iff there is some state

$$\lhd\,\mathbf{a}\vdash \epsilon\left(s \,\middle\|\, \lhd\,\mathbf{a}'\vdash \epsilon:\pi\right) \in S.$$

Note that a given process in some state $s \in S$ can cause several different tuples to appear in $\alpha_1(S)$, one for each point in the control path of that process at which a spawn occurred. This is a key idea: that we will be focusing on the branching points.

After invoking an isomorphism on the codomain of α_1, we have a representation of a set of states by

$$CP \rightarrow (Side \times CP) \rightarrow \wp(Frame)$$

where $Side = \{\lhd, \rhd\}$ and the domain corresponds to the first component of the codomain of α_1. Note that we must indeed have $\wp(Frame)$ instead of $Frame$ here because two different states in S may each have a process with the same control path, but with different frames due to communication with other processes.

Next comes the partitioning of control paths. This is done by the pointwise application of α_2, which yields

$$CP \rightarrow ((Side \times \widehat{CP}) \times Addr) \rightarrow \wp(Frame).$$

For each control path $\mathbf{a}$, α_2 applies the function *partition* to the path and also maintains the final program address of the path. Formally, $\pi \in \dot{\alpha}_2(\alpha_1(S))(\mathbf{a})((\lhd, \hat{\mathbf{a}}), a)$ iff there is some state

$$\lhd\!\boxed{\mathbf{a}} \!-\epsilon \left(\lhd\!\boxed{\mathbf{a}'}\!-a:\pi \;\Big\|\; s \right) \in S$$

and $partition(\mathbf{a}'a) = \hat{\mathbf{a}}$; similarly for $\rhd$, with the branches reversed.

Maintaining the final program address is analogous to the standard application of abstract interpretation to find local invariants, or collected data at each syntactic point in the program. Keeping the partitioning is similar to techniques that generally fall under the name *polyvariance*; not dissimilar are Harrison's control strings [Har89]. However, polyvariance alone yields only a uniform analysis. The chief interest in our abstraction lies not in each individual (polyvariant) function in $\dot{\alpha}_2(\alpha_1(S))$, but rather in the mapping from control paths to these functions. Indeed, the abstraction in the next part will approximate these functions differently, relative to the control paths that map to them, and it is this technique that achieves the non-uniformity. The advantage of this will become clear in Section 6.

Since at each step we have a complete lattice under the pointwise subset ordering $\dot{\subseteq}$, then as in [CC92] the abstraction functions α_1 and α_2 define associated concretization functions γ_1 and γ_2, as shown in Figure 3.

5.2 Abstracting values relative to their ancestor spawns

At this point, the most complicated step of the abstraction is applied: the approximation of a set of frames with a member of the set

$$\widehat{Frame} = Var + \{\mathrm{K}\} \rightarrow \wp(Source \times Addr)$$

where $Source = Branch + \{\text{EXTERNAL}\}$ and $Branch = Side \times \widehat{CP}$. An abstract frame $\hat{\pi} \in \widehat{Frame}$ assigns a set of abstract values $\hat{v}$ to each variable and to the extra "variable" K representing the continuation. The recursively-defined function $[\![\;]\!]^{\mathbf{a}}_X$ assigns to abstract frames and values their concrete counterparts under $X \in \widehat{State}$ and control path $\mathbf{a}$.

$\hat{\pi} \in \widehat{Frame}$	$[\![\hat{\pi}]\!]^{\mathbf{a}}_X \in \wp(Frame + Env)$
$\widehat{V} \in \wp(Source \times Addr)$	$[\![\widehat{V}]\!]^{\mathbf{a}}_X \in \wp(Val)$
$\hat{v} \in Source \times Addr$	$[\![\hat{v}]\!]^{\mathbf{a}}_X \in \wp(Val)$

Its definition is given in Figure 3.

Note that the domain of $\widehat{State}$ is $SAddr$, the set of program addresses of spawn expressions, but the domain of the previous step in the abstraction (the codomain of γ_3) is CP. This step of the abstraction strips off all but the final program address of this control path, which is a program address of a spawn expression. Therefore, given $X \in \widehat{State}$ and $a_s \in SAddr$, $X(a_s)$ represents the pairs of processes that split at one of possibly many different control paths $\mathbf{a}a_s$, as described in the introduction to Section 5. But the subtle point is that the meaning that γ_3 gives to abstract channel values in $\hat{\pi} = X(a_s)(b, a)$ *is relative to the entire control path* $\mathbf{a}a_s$, not just a_s. This is why $[\![\;]\!]^{\mathbf{a}}_X$ is parameterized by a control path. Recall that a channel value is the control path at which it was created. The possible abstract channel values are, for $P@a = \mathsf{channel}\ e$:

- $((\lhd, \hat{\mathbf{a}}), a)$, representing a control path with prefix $\mathbf{a}a_s$ that then takes the left side of the spawn at a_s (*i.e.*, the side beginning with a_s.proc) and continues through a control path that is in partition $\hat{\mathbf{a}}$ and ends with a,
- $((\rhd, \hat{\mathbf{a}}), a)$, representing a control path with prefix $\mathbf{a}a_s$ that then takes the right side of the spawn at a_s (*i.e.*, the side beginning with a_s) and continues through a control path that is in partition $\hat{\mathbf{a}}$ and ends with a, and
- $(\text{EXTERNAL}, a)$, representing a control path that ends with a and does not have $\mathbf{a}a_s$ as a prefix.

Channel values are the only ones that directly use the control path $\mathbf{a}$, but the meanings of other types of values use the same technique. In general, the meaning of an abstract value (q, a) is relative to a control path $\mathbf{a}$ that ends with a spawn; it represents values that were created at a control path that ends with a and either

- begins with $\mathbf{a}$, takes the left branch, and whose remainder after the spawn is in partition $\hat{\mathbf{a}}$ (if $q = (\lhd, \hat{\mathbf{a}})$),
- begins with $\mathbf{a}$, takes the right branch, and whose remainder after the spawn is in partition $\hat{\mathbf{a}}$ (if $q = (\rhd, \hat{\mathbf{a}})$), or
- did not pass through this spawn (if $q = \text{EXTERNAL}$).

The last is called EXTERNAL because intuitively it corresponds to values that exist in environments within the subtree $\mathbf{a}$, but that were created outside that subtree, either inherited from an ancestor process or received from another process.

If an abstract value represents a value with subcomponents (*i.e.*, records, constructed data, and closures), then the values of those subcomponents are derived from the environment at their creation point[2]. This is why X itself is a parameter of $[\![\;]\!]^{\mathbf{a}}_X$. For instance, if $P@a = c\ e$, then $[\![(q, a)]\!]^{\mathbf{a}a_s}_X$ represents values $c\ v$ that were created at a and at point q relative to $\mathbf{a}a_s$. If $q \in Side$, then we can just look up $X(a_s)(q, a)(\langle a.\mathrm{arg}\rangle)$ directly to retrieve the abstract values for v. But if $q = \text{EXTERNAL}$, then it could have come from any control path ending with a, and hence the definition at the bottom of Figure 3.

The reason why there is only a concretization function γ_3 for the last step, and no associated abstraction function α_3, is that a value does not have a unique abstract representation in $\widehat{Frame}$. For instance, given an constant k, there may be more than one occurrence of the expression k in the program, and so abstract values with these different program addresses may concretize to the same value[3]. From [CC92], the function γ_3 is sufficient for an abstract interpretation because $\widehat{State}$ is a complete lattice.

[2]This is a technique used in some closure analyses (*e.g.*[Hei92])

[3]More precisely, $\widehat{State}$ is an abstraction of a more expressive set of states that maintains information about the creation points of values That formulation of the standard semantics would be useful for, *e.g.*, a general alias analysis [Col].

Given a function $f \in A \rightarrow B$, we write $\dot{f}$ for the pointwise extension of f, we write $\wp(f)$ for the function $\lambda x.\{f(a) \mid a \in x\}$ in $\wp(A) \rightarrow \wp(B)$, and we write f^{-1} for the inverse image of f (a function in $B \rightarrow \wp(A)$). We implicitly use tuple isomorphisms such as $A \times (B \times C) \cong A \times B \times C$. The approximation of $\wp(State)$ with $\widehat{State}$ by $\gamma = \gamma_1 \circ \dot{\gamma}_2 \circ \gamma_3$ is:

$$\begin{array}{llcl}
\wp(State) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} & \wp(CP \times Side \times CP \times Frame) & \cong & CP \rightarrow (Side \times CP) \rightarrow \wp(Frame) \\
 & & \underset{\dot{\alpha}_2}{\overset{\dot{\gamma}_2}{\leftrightarrows}} & CP \rightarrow (Branch \times Addr) \rightarrow \wp(Frame) \\
 & & \overset{\gamma_3}{\longleftrightarrow} & SAddr \rightarrow (Branch \times Addr) \rightarrow \widehat{Frame} \quad = \quad \widehat{State}
\end{array}$$

with

$$\begin{array}{lllll}
z \in Side & = & \{\triangleleft, \triangleright\} & & \\
b \in Branch & = & Side \times \widehat{CP} & SAddr & = \{a \in Addr \mid P@a = \texttt{spawn} \ldots\} \\
q \in Source & = & Branch + \{\text{EXTERNAL}\} & \hat{\pi} \in \widehat{Frame} & = Var + \{\mathrm{K}\} \rightarrow \wp(Source \times Addr)
\end{array}$$

where α_1 and α_2 are defined as follows ($\gamma_i(y) = \dot{\cup}\{x \mid \alpha_i(x) \dot{\subseteq} y\}$ for $i \in \{1, 2\}$):

$$\begin{array}{lcll}
\alpha_1 & = & \cup \circ \wp(pairs) & \in \wp(State) \rightarrow \wp(CP \times Side \times CP \times Frame) \\
\alpha_2 & = & \lambda f.(\cup \circ \wp(f) \circ branch^{-1}) & \in ((Side \times CP) \rightarrow \wp(Frame)) \rightarrow (Branch \times Addr) \rightarrow \wp(Frame)
\end{array}$$

$$\begin{array}{rcl}
pairs & \in & State \rightarrow \wp(CP \times Side \times CP \times Frame) \\
pairs(\mathbf{a}\!:\!\pi) & = & \emptyset \\
pairs\left(\mathbf{a}\left(s_1 \,\middle\|\, s_2\right)\right) & = & (\{(\mathbf{a}, \triangleleft)\} \times procs(s_1)) \cup (\{(\mathbf{a}, \triangleright)\} \times procs(s_2)) \cup pairs(\mathbf{a}s_1) \cup pairs(\mathbf{a}s_2)
\end{array}$$

$$\begin{array}{rcl}
procs & \in & State \rightarrow \wp(CP \times Frame) \\
procs(\mathbf{a}\!:\!\pi) & = & \{(\mathbf{a}, \pi)\} \\
procs\left(\mathbf{a}\left(s_1 \,\middle\|\, s_2\right)\right) & = & procs(\mathbf{a}s_1) \cup procs(\mathbf{a}s_2)
\end{array}$$

$$\begin{array}{rcl}
branch & \in & (Side \times CP) \rightarrow (Branch \times Addr) \\
branch(z, \mathbf{a}a) & = & ((z, partition(\mathbf{a}a)), a)
\end{array}$$

and where $\gamma_3 \in (SAddr \rightarrow (Branch \times Addr) \rightarrow \widehat{Frame}) \rightarrow CP \rightarrow (Branch \times Addr) \rightarrow \wp(Frame)$ is defined as follows:

$$\gamma_3(X)(\mathbf{a}a_s)(b, a) = \{\pi \in [\![X(a_s)(b, a)]\!]_X^{\mathbf{a}a_s}\}$$

$$\begin{array}{rcll}
(\rho, \bullet) & \in & [\![\hat{\pi}]\!]_X^{\mathbf{a}} & \text{iff} \quad \rho \in [\![\hat{\pi}]\!]_X^{\mathbf{a}} \\
(\rho, a\!:\!\pi) & \in & [\![\hat{\pi}]\!]_X^{\mathbf{a}a_s} & \text{iff} \quad \rho \in [\![\hat{\pi}]\!]_X^{\mathbf{a}a_s} \wedge \exists q \in Source.((q, a) \in \hat{\pi}(\mathrm{K}) \wedge \pi \in [\![X(a_s)(q, a)]\!]_X^{\mathbf{a}a_s}) \\
 & & & \\
\rho & \in & [\![\hat{\pi}]\!]_X^{\mathbf{a}} & \text{iff} \quad \forall x \in Var.\, \rho(x) \text{ defined} \Rightarrow \rho(x) \in [\![\hat{\pi}(x)]\!]_X^{\mathbf{a}} \\
 & & & \\
v & \in & [\![\widehat{V}]\!]_X^{\mathbf{a}} & \text{iff} \quad \exists \hat{v} \in \widehat{V}.\, v \in [\![\hat{v}]\!]_X^{\mathbf{a}} \\
 & & & \\
k & \in & [\![(q, a)]\!]_X^{\mathbf{a}} & \text{iff} \quad P@a = k \\
\{\} & \in & [\![(q, a)]\!]_X^{\mathbf{a}} & \text{iff} \quad P@a = \{\} \vee P@a = \texttt{spawn}\ e \vee P@a = \texttt{transmit}\ e \\
\{f_1 = v_1, \ldots, f_n = v_n\} & \in & [\![(q, a)]\!]_X^{\mathbf{a}a_s} & \text{iff} \quad P@a = \{f_1\texttt{=}e_1, \ldots, f_n\texttt{=}e_n\} \wedge \forall 1 \le i \le n.\, v_i \in [\![X(a_s)(q, a)(\langle a.\mathrm{field}_i \rangle)]\!]_X^{\mathbf{a}a_s} \\
c\ v & \in & [\![(q, a)]\!]_X^{\mathbf{a}a_s} & \text{iff} \quad P@a = c\ e \wedge v \in [\![X(a_s)(q, a)(\langle a.\mathrm{arg} \rangle)]\!]_X^{\mathbf{a}a_s} \\
(a, \rho) & \in & [\![(q, a)]\!]_X^{\mathbf{a}a_s} & \text{iff} \quad (P@a = \texttt{fn}\ x\ \texttt{=>}\ e \vee P@a = \texttt{rec}\ y(x)\ \texttt{=}\ e) \wedge \rho \in [\![X(a_s)(q, a)]\!]_X^{\mathbf{a}a_s} \\
\mathbf{a}a_s(a_s.\mathrm{proc})\mathbf{a}'a & \in & [\![((\triangleleft, \hat{\mathbf{a}}), a)]\!]_X^{\mathbf{a}a_s} & \text{iff} \quad P@a = \texttt{channel}\ e \wedge partition((a_s.\mathrm{proc})\mathbf{a}'a) = \hat{\mathbf{a}} \\
\mathbf{a}a_s a_s \mathbf{a}'a & \in & [\![((\triangleright, \hat{\mathbf{a}}), a)]\!]_X^{\mathbf{a}a_s} & \text{iff} \quad P@a = \texttt{channel}\ e \wedge partition(a_s\mathbf{a}'a) = \hat{\mathbf{a}} \\
\mathbf{a}'a & \in & [\![(\text{EXTERNAL}, a)]\!]_X^{\mathbf{a}} & \text{iff} \quad P@a = \texttt{channel}\ e \wedge \mathbf{a} \text{ is not a prefix of } \mathbf{a}'
\end{array}$$

$$X(a_s)(\text{EXTERNAL}, a)(x) = \cup\{source(X(a_s')(b, a)(x)) \mid a_s' \in SAddr \wedge b \in Branch\}$$

$$source(\widehat{V}) = \{(q, a) \mid q \in Source \wedge \exists (q', a) \in \widehat{V}\}$$

Figure 3: The approximation of sets of states with $\widehat{State}$, parameterized by $\widehat{CP}$ and $partition \in CP \rightarrow \widehat{CP}$.

6 An Application of the Approximation

We now formally present the main motivation for the design of the $\widehat{State}$ approximation: a way to determine pairs of dynamic occurrences of channels that are unequal.

Proposition 1 *For $\widehat{V_1}, \widehat{V_2} \in \wp(Source \times Addr)$ we define $\widehat{V_1} \equiv \widehat{V_2}$ iff there exists $(q, a) \in \widehat{V_1} \cap \widehat{V_2}$ such that $P@a =$ channel e. Then if the following three conditions hold for any $X \in \widehat{State}$:*

$$X(a_s)((\triangleleft, \hat{\mathbf{a}}_1), a_1)(x_1) \not\equiv X(a_s)((\triangleright, \hat{\mathbf{a}}_2), a_2)(x_2)$$

$$\lhd\,\mathbf{a}\vdash a_s\left(\lhd\,\mathbf{a}_1\vdash a_1 : (\rho_1, \kappa_1) \;\middle\|\; \lhd\,\mathbf{a}_2\vdash a_2 : (\rho_2, \kappa_2)\right) \in \gamma(X)$$

$$partition(\mathbf{a}_1 a_1) = \hat{\mathbf{a}}_1 \;\wedge\; partition(\mathbf{a}_2 a_2) = \hat{\mathbf{a}}_2$$

then $\rho_1(x_1)$ and $\rho(x_2)$ are not the same channel.

This proposition is useful when X is the least fixpoint of the abstract transition function presented in the next section, because then "$\ldots \in \gamma(X)$" becomes "$\ldots$ is a state reachable in some evaluation of P". Then this proposition becomes a powerful automatic proof technique for determining the communication pattern of P because it eliminates many possibilities for synchronizations that cannot happen.

The polyvariance achieved by the control-path partitioning $\widehat{CP}$ is *not* the source of the non-uniformity in the analysis; rather, it is the universal quantification over $\mathbf{a}$ in this proposition that allows an element of $\widehat{State}$ to distinguish between iterations in an infinite recursive pattern. Hence our comments at the end of Section 5.1. In Section 8 we given an example.

7 The Abstract Transition Function

With the design in the previous section of an expressive finite approximation of a set of states, we completed the major conceptual task of the algorithm design. But we still need an abstract transition function that operates on this representation that is safe with respect to the semantics. The transition function of the semantics maps sets of states to sets of states:

$$T(S) = \{s' \mid s' = s_{init} \;\vee\; \exists s \in S.\, s \Longrightarrow s'\}$$

We need a function

$$\widehat{T} \in \widehat{State} \rightharpoonup \widehat{State}$$

such that $T \circ \gamma \mathrel{\dot{\subseteq}} \gamma \circ \widehat{T}$. By [CC77], we then have that the least fixpoint of $\widehat{T}$ represents a superset of all states s such that $s_{init} \Longrightarrow^{*} s$. This fixpoint, which can be obtained for instance by standard iteration techniques, is the result of the analysis.

The abstract transition function $\widehat{T}$ is defined componentwise; each abstract frame $\widehat{T}(X)(a_s)(b, a)$ is defined with respect to the abstract frames of X by a series of rules that mirror the rules of the semantics in Figures 1 and 2. All rules for $\widehat{T}$ except those that are related to synchronization are shown in Figure 4, and the rules for synchronization are given in Figure 5.

As an example, consider the rule for constants. If $P@a = k$, then:

$$X(a_s)(b, a)\{\!\!\{\langle a\rangle \mapsto \{(b, a)\}\}\!\!\} \mathrel{\dot{\subseteq}} Y(a_s)(b, a)$$

This corresponds to the semantic rule:

$$\lhd\,\mathbf{a}\vdash \overline{a} : (\rho, \kappa) \Longrightarrow \lhd\,\mathbf{a}\vdash aa : (\rho[\langle a\rangle \mapsto k], \kappa)$$

In particular, if we assume for instance that $b = (\triangleleft, \hat{\mathbf{a}})$, it corresponds to the following subclass of this rule, where $\mathbf{a} = \mathbf{a}' a_s \mathbf{a}''$ and $partition(\mathbf{a}'' a) = \hat{\mathbf{a}}$:

$$\begin{aligned} &\lhd\,\mathbf{a}'\vdash a_s\left(\lhd\,\mathbf{a}''\vdash \overline{a} : (\rho, \kappa) \;\middle\|\; s\right) \\ \Longrightarrow\;& \lhd\,\mathbf{a}'\vdash a_s\left(\lhd\,\mathbf{a}''\vdash aa : (\rho[\langle a\rangle \mapsto k], \kappa) \;\middle\|\; s\right)\end{aligned}$$

(If instead $b = (\triangleright, \hat{\mathbf{a}})$ then the branches would swap.) In this case, $Y(a_s)(b, a)$ is short for $\widehat{T}(X)(a_s)((\triangleleft, \hat{\mathbf{a}} \oplus a), a)$, and so we expect that $partition(\mathbf{a}'' aa) = \hat{\mathbf{a}} \oplus a$. In general $\oplus$ must satisfy the property that

$$partition(\mathbf{a}) \oplus a = partition(\mathbf{a}a).$$

So the rule of $\widehat{T}$ for constants describes how to define the new frame from the old one, which is simply a new binding of an abstract constant created at branch b terminating with program address a.

Most of the remaining rules are derived the corresponding rules of the standard semantics in the same manner. The exceptions to this are the last two rules for process spawn and the rules for synchronization. Consider the following subclass of the spawn rule of the standard semantics, where $\pi_1 = (\rho, \bullet)$ and $\pi_2 = (\rho[\langle a\rangle \mapsto \{\}], \kappa)$:

$$\begin{aligned} &\lhd\,\mathbf{a}\vdash a_s\left(\lhd\,\mathbf{a}'\vdash \overline{a} : (\rho, \kappa) \;\middle\|\; s\right) \\ \Longrightarrow\;& \lhd\,\mathbf{a}\vdash a_s\left(\lhd\,\mathbf{a}'\vdash a\left(a.\text{proc} : \pi_1 \;\middle\|\; a : \pi_2\right) \;\middle\|\; s\right) \\ =\;& \lhd\,\mathbf{a}a_s\mathbf{a}'\vdash a\left(\lhd\,\epsilon\vdash a.\text{proc} : \pi_1 \;\middle\|\; \lhd\,\epsilon\vdash a : \pi_2\right)\end{aligned}$$

The middle line corresponds to the first two spawn rules of $\widehat{T}$, where $b = (\triangleleft, \hat{\mathbf{a}})$ and $partition(\mathbf{a}' a) = \hat{\mathbf{a}}$:

$$\begin{aligned} X(a_s)(b, a)\{\!\!\{\mathrm{K} \mapsto \emptyset\}\!\!\} &\mathrel{\dot{\subseteq}} Y(a_s)(b, a.\text{proc}) \\ X(a_s)(b, a)\{\!\!\{\langle a\rangle \mapsto \{(b, a)\}\}\!\!\} &\mathrel{\dot{\subseteq}} Y(a_s)(b, a)\end{aligned}$$

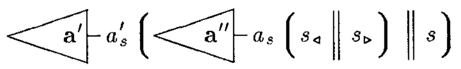
But the last line in the above transition corresponds to an alternate view of the right-hand-side state—a view from the new branch at a. This corresponds to a completely different entry in X for the same state, so π_1 and π_2 have to be "copied" to that entry. This is done with the remaining two spawn rules:

$$\begin{aligned} external(X(a_s)(b, a)\{\!\!\{\mathrm{K} \mapsto \emptyset\}\!\!\}) &\mathrel{\dot{\subseteq}} Y(a)((\triangleleft, \hat{\epsilon}), a.\text{proc}) \\ external(X(a_s)(b, a)\{\!\!\{\langle a\rangle \mapsto \{(b, a)\}\}\!\!\}) & \\ &\mathrel{\dot{\subseteq}} Y(a)((\triangleright, \hat{\epsilon}), a)\end{aligned}$$

Here, $\hat{\epsilon} \in \widehat{CP}$ is $partition(\epsilon)$, corresponding to the occurrences of ϵ in the new branch above, and the source of all values in the environment must change to EXTERNAL because they are inherited from outside the new branch.

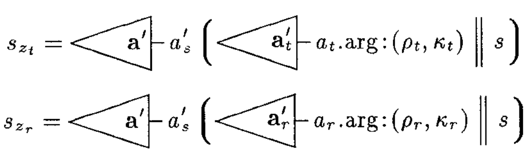
The rules for synchronization are given in Figure 5. The first two rules exactly match the synchronization rule of the standard semantics as written in Figure 2, where $\hat{\mathbf{a}}_t = partition(\mathbf{a}_t(a.\text{arg}))$ and $\hat{\mathbf{a}}_r = partition(\mathbf{a}_r(a.\text{arg}))$.

This is recording a synchronization between processes at the point a_s at which they branched. But this synchronization is also visible when the state is viewed from different spawn points. The middle two synchronization rules cover the effect of this communication on a view from a branch at a'_s that occurred *before* the branch at a_s. In other words, $\mathbf{a} = \mathbf{a}'a'_s\mathbf{a}''$ for some $\mathbf{a}'$ and $\mathbf{a}''$, and $b = (z, partition(\mathbf{a}''a_s))$. If for instance $z = \triangleleft$, then the class of states covered are:



And the branches b_t and b_r, into s_{z_t} and s_{z_r} respectively, need to be prefixed with b by the operator $\otimes$.

The last two rules for synchronization handle branching points *after* the branch at a_s, within s_{z_t} and s_{z_r} respectively. If $b = (z, partition(\mathbf{a}'a'_s))$, $b'_t = (z'_t, partition(\mathbf{a}'_t(a_t.\text{arg})))$, and $b'_r = (z'_r, partition(\mathbf{a}'_r(a_r.\text{arg})))$, then the class of states covered in each case where *e.g.* $z'_t = \triangleleft$ and $z'_r = \triangleleft$ are:



where $\mathbf{a}'a'_s\mathbf{a}'_t = \mathbf{a}_t$ and $b = b_t$ or $\mathbf{a}'a'_s\mathbf{a}'_r = \mathbf{a}_r$ and $b = b_r$, respectively. The latter case, for the `receive` expression, is troublesome because the value received comes from a transmit that is not within the subtree at branch a'_s. However, EXTERNAL is not sufficient to model such values, since a value may have been created within that subtree, transmitted outside, and then transmitted to the receive currently being considered. Therefore, we must assume any member of *Source* for the received value.

8 An Example

We now present the `fold` example of Section 1. Let us assume that `fold` is one function (written with syntactic sugar for sequencing, let bindings, and pattern-matching for pair arguments) in an otherwise arbitrary program P. We must first choose an appropriate partition $\widehat{CP}$ of CP. We pick a simple one that will suffice for this example:

$$\begin{aligned} \widehat{CP} &= \{\text{ORIG}, \text{NEW}\} \\ partition(\mathbf{a}) &= \begin{cases} \text{NEW} & \text{if } \exists \mathbf{a}' a_s (a_s.\text{proc})\mathbf{a}'' = \mathbf{a} \\ \text{ORIG} & \text{otherwise} \end{cases} \end{aligned}$$

CP is partitioned into two classes: the class NEW for control paths that have gone through the left side of some spawn branch (*i.e.*, those that terminate in a different process from their beginning) and the class ORIG for all other control paths (*i.e.*, those that remain in the same process from beginning to end). As explained in Section 6, $\widehat{CP}$ is not the source of the precision of this analysis, and simple choices will typically suffice.

We also need to define the following operations for the abstract transition function $\widehat{T}$:

$$\begin{aligned} \hat{\epsilon} &= \text{ORIG} \\ \hat{\mathbf{a}} \oplus a &= \begin{cases} \text{NEW} & \text{if } a = a'.\text{proc for some } a' \\ \hat{\mathbf{a}} & \text{otherwise} \end{cases} \\ \hat{\mathbf{a}} \otimes b &= \begin{cases} \text{ORIG} & \text{if } \hat{\mathbf{a}} = \text{ORIG} \wedge b = (\triangleright, \text{ORIG}) \\ \text{NEW} & \text{otherwise} \end{cases} \end{aligned}$$

We then calculate the least fixpoint Z of $\widehat{T}$, which represents all possible traces of an evaluation of P. The information in Z relevant to determining the communication topology of `fold` is summarized in Table 1. This information is independent of the rest of the program, including the process creation and communication behavior of functions `combine` and `base`, and including the possibility that those functions and/or other parts of the program may call `fold` anywhere. An entry in the table at row $[a_s\ z\ \hat{\mathbf{a}}]$ and column $x\,@\,a$ is $Z(a_s)((z, \hat{\mathbf{a}}), a)(x)$, the set of abstractions of all values that were bound to x in some process during evaluation of P that is at program address a, whose control path took either the left side (new process) of some spawn branch at program address a_s if $z = \triangleleft$ or the right side (existing process) if $z = \triangleright$, and whose control-path suffix since that point was in partition $\hat{\mathbf{a}}$.

The first column shows the values of the channel component of the arguments to the two `transmit` expressions (they are the same). The two other columns show the values of the (channel) argument of the two `receive` expressions.

For the values, $\text{ORIG}^{\triangleleft}$ is short for $(\triangleleft, \text{ORIG})$, and similarly for $\text{NEW}^{\triangleleft}$, $\text{ORIG}^{\triangleright}$, and $\text{NEW}^{\triangleright}$. Each occurrence of the token $\widehat{V}_?$ stands for some set that is not shown, but does not include $(q, \boxed{L})$ or $(q, \boxed{R})$ for any $q \in Source$. The a_{other} entries are valid for all program addresses of spawn expressions in P other than $\boxed{SL}$ and $\boxed{SR}$.

Consider the entry in the first column of row $[\boxed{SL}\ \triangleleft\ \text{NEW}]$. This column represents the possible values of the channel of each transmit expression in some process that whose control path went through the left side (*i.e.*, new process) of a spawn branch at $\boxed{SL}$ (from the $\triangleleft$) and whose control-path suffix since that point included a left side of another spawn branch, possibly also at $\boxed{SL}$ (from the NEW). Now consider the abstract value $(\text{ORIG}^{\triangleleft}, \boxed{L})$ in this entry. It represents channels created at program address $\boxed{L}$ in the process whose control path also went through the left side of this *same* spawn branch (from the $\triangleleft$), but whose control-path suffix since that point did not include a left side of another spawn branch (from the ORIG). In contrast, the abstract value $(\text{NEW}^{\triangleleft}, \boxed{L})$ represents channels created at program address $\boxed{L}$ in a process whose control path also went through the left side of this spawn branch and whose control-path suffix since that point included a left side of another spawn branch (but not necessarily the same branch that caused the NEW in this row index).

We can apply Proposition 1 to this table to determine the communication topology of the program. Many pairs of entries in this table match with the relation $\equiv$, but according to the proposition the only pairs we need to consider are ones comprising a transmit in some row $[a_s\ z_t\ \hat{\mathbf{a}}_t]$ and a receive in some row $[a_s\ z_r\ \hat{\mathbf{a}}_r]$ such that $z_t \neq z_r$. There are only two such pairs that match, and the witnesses to those matches are boxed in Table 1. The *only* possible synchro-

The definition of $\widehat{T}(X)$ is given componentwise below, where $Y(a_s)((z,\hat{\mathrm{a}}),a) \stackrel{\mathrm{def}}{=} \widehat{T}(X)(a_s)((z,\hat{\mathrm{a}}\oplus a),a)$.

$$\mathit{reach}(\hat{\pi}) = \exists x.\, \hat{\pi}(x) \neq \emptyset$$

$$X(a_s)(b,a)(x.c) = \cup\{X(a_s)(q,a')(\langle a'.\mathrm{arg}\rangle) \mid (q,a') \in X(a_s)(b,a)(x) \ \wedge\ P@a' = c\ e\}$$

$$X(a_s)(b,a)(x.f) = \cup\{X(a_s)(q,a')(\langle a'.\mathrm{field}_i\rangle) \mid (q,a') \in X(a_s)(b,a)(x) \ \wedge\ P@a' = \{f_1\texttt{=}e_1\texttt{,}\ldots\texttt{,}f_n\texttt{=}e_n\} \ \wedge\ f = f_i\}$$

$$\mathit{external}(\hat{\pi})(x) = \{(\textsc{external},a) \mid \exists (q,a) \in \hat{\pi}(x)\}$$

$$\hat{\pi}(\!|x \mapsto A|\!) = \begin{cases} \hat{\pi}[x \mapsto A] & \text{if } \mathit{reach}(\hat{\pi}) \\ \lambda x.\,\emptyset & \text{otherwise} \end{cases}$$

$$\begin{array}{rcll}
X(a_s)(b,a)(\!|\langle a\rangle \mapsto \{(b,a)\}|\!) & \dot\subseteq & Y(a_s)(b,a) & \text{if } P@a = k \\
\\
X(a_s)(b,a)(\!|\langle a\rangle \mapsto X(a_s)(b,a)(x)|\!) & \dot\subseteq & Y(a_s)(b,a) & \text{if } P@a = x \\
\\
X(a_s)(b,a)(\!|\langle a\rangle \mapsto \{(b,a)\}|\!) & \dot\subseteq & Y(a_s)(b,a) & \text{if } P@a = \{\} \\
X(a_s)(b,a) & \dot\subseteq & Y(a_s)(b,a.\mathrm{field}_1) & \text{if } P@a = \{f_1\texttt{=}e_1\texttt{,}\ldots\texttt{,}f_n\texttt{=}e_n\} \\
X(a_s)(b,a.\mathrm{field}_i) & \dot\subseteq & Y(a_s)(b,a.\mathrm{field}_{i+1}) & \text{if } P@a = \{f_1\texttt{=}e_1\texttt{,}\ldots\texttt{,}f_n\texttt{=}e_n\} \ \wedge\ 1 \le i < n \\
X(a_s)(b,a.\mathrm{field}_n)(\!|\langle a\rangle \mapsto \{(b,a)\}|\!) & \dot\subseteq & Y(a_s)(b,a) & \text{if } P@a = \{f_1\texttt{=}e_1\texttt{,}\ldots\texttt{,}f_n\texttt{=}e_n\} \\
\\
X(a_s)(b,a) & \dot\subseteq & Y(a_s)(b,a.\mathrm{pred}) & \text{if } P@a = \texttt{case}\ldots \\
X(a_s)(b,a.\mathrm{pred})(\!|\langle a\rangle \mapsto X(a_s)(b,a)(\langle a.\mathrm{pred}\rangle.c_i)|\!) & \dot\subseteq & Y(a_s)(b,a.\mathrm{case}_i) & \text{if } P@a = \texttt{case}\ e\ \texttt{of}\ \ldots\ \texttt{|}\ c_i\,x_i\ \texttt{=>}\,e_i\ \texttt{|}\ \ldots \\
X(a_s)(b,a.\mathrm{case}_i)(\!|\langle a\rangle \mapsto X(a_s)(b,a)(\langle a.\mathrm{case}_i\rangle)|\!) & \dot\subseteq & Y(a_s)(b,a) & \\
\\
X(a_s)(b,a) & \dot\subseteq & Y(a_s)(b,a.\mathrm{arg}) & \text{if } P@a = \mathit{op}\ e \\
X(a_s)(b,a.\mathrm{arg})(\!|\langle a\rangle \mapsto \{(b,a)\}|\!) & \dot\subseteq & Y(a_s)(b,a) & \text{if } P@a = c\ e \\
X(a_s)(b,a.\mathrm{arg})(\!|\langle a\rangle \mapsto X(a_s)(b,a.\mathrm{arg})(\langle a.\mathrm{arg}\rangle.f)|\!) & \dot\subseteq & Y(a_s)(b,a) & \text{if } P@a = \texttt{\#}f\ e \\
\\
X(a_s)(b,a)(\!|\langle a\rangle \mapsto \{(b,a)\}|\!) & \dot\subseteq & Y(a_s)(b,a) & \text{if } P@a = \texttt{fn}\ x\ \texttt{=>}\ e \ \vee\ P@a = \texttt{rec}\ y(x)\ \texttt{=}\ e \\
\\
X(a_s)(b,a) & \dot\subseteq & Y(a_s)(b,a.\mathrm{op}) & \text{if } P@a = e\ e' \\
X(a_s)(b,a.\mathrm{op}) & \dot\subseteq & Y(a_s)(b,a.\mathrm{arg}) & \\
\multicolumn{4}{l}{X(a_s)(q,a')(\!|x \mapsto X(a_s)(b,a.\mathrm{arg})(\langle a.\mathrm{arg}\rangle)|\!)(\!|\mathrm{K} \mapsto \{(b,a)\}|\!)} \\
 & \dot\subseteq & Y(a_s)(b,a'.\mathrm{body}) & \text{if } P@a = e\ e' \ \wedge\ P@a' = \texttt{fn}\ x\ \texttt{=>}\ e'' \\
 & & & \quad \wedge\ (q,a') \in X(a_s)(b,a.\mathrm{arg})(\langle a.\mathrm{op}\rangle) \\
\multicolumn{4}{l}{X(a_s)(q,a')(\!|x \mapsto X(a_s)(b,a.\mathrm{arg})(\langle a.\mathrm{arg}\rangle)|\!)(\!|y \mapsto \{(q,a')\}|\!)(\!|\mathrm{K} \mapsto \{(b,a)\}|\!)} \\
 & \dot\subseteq & Y(a_s)(b,a'.\mathrm{body}) & \text{if } P@a = e\ e' \ \wedge\ P@a' = \texttt{rec}\ y(x)\ \texttt{=}\ e'' \\
 & & & \quad \wedge\ (q,a') \in X(a_s)(b,a.\mathrm{arg})(\langle a.\mathrm{op}\rangle) \\
X(a_s)(q,a')(\!|\langle a'\rangle \mapsto X(a_s)(b,a.\mathrm{body})(\langle a.\mathrm{body}\rangle)|\!) & \dot\subseteq & Y(a_s)(b,a') & \text{if } (q,a') \in X(a_s)(b,a.\mathrm{body})(\mathrm{K}) \\
\\
X(a_s)(b,a.\mathrm{arg})(\!|\langle a\rangle \mapsto \{(b,a)\}|\!) & \dot\subseteq & Y(a_s)(b,a) & \text{if } P@a = \texttt{channel}\ e \\
\\
X(a_s)(b,a)(\!|\mathrm{K} \mapsto \emptyset|\!) & \dot\subseteq & Y(a_s)(b,a.\mathrm{proc}) & \text{if } P@a = \texttt{spawn}\ e \\
X(a_s)(b,a)(\!|\langle a\rangle \mapsto \{(b,a)\}|\!) & \dot\subseteq & Y(a_s)(b,a) & \text{if } P@a = \texttt{spawn}\ e \\
\mathit{external}(X(a_s)(b,a)(\!|\mathrm{K} \mapsto \emptyset|\!)) & \dot\subseteq & Y(a)((\triangleleft,\hat{\epsilon}),a.\mathrm{proc}) & \text{if } P@a = \texttt{spawn}\ e \\
\mathit{external}(X(a_s)(b,a)(\!|\langle a\rangle \mapsto \{(b,a)\}|\!)) & \dot\subseteq & Y(a)((\triangleright,\hat{\epsilon}),a) & \text{if } P@a = \texttt{spawn}\ e
\end{array}$$

Figure 4: The abstract transition function $\widehat{T}$ without synchronization, parameterized by $\hat{\epsilon} \in \widehat{CP}$ and $\oplus \in \widehat{CP} \times \mathit{Addr} \to \widehat{CP}$.

If

$$P@a_t = \texttt{transmit}\ e \qquad P@a_r = \texttt{receive}\ e'$$

$$b_t = (z_t, \hat{\mathbf{a}}_t) \qquad b_r = (z_r, \hat{\mathbf{a}}_r) \qquad z_t \neq z_r$$

$$X(a_s)(b_t, a_t.\mathrm{arg})(\langle a_t.\mathrm{arg}\rangle.1) \equiv X(a_s)(b_r, a_r.\mathrm{arg})(\langle a_r.\mathrm{arg}\rangle)$$

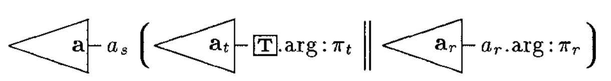
then

$$\begin{array}{rcll}
X(a_s)(b_t, a_t.\mathrm{arg})[\![\langle a_t\rangle \mapsto \{(b_t, a_t)\}]\!] & \dot{\subseteq} & Y(a_s)(b_t, a_t) & \\
X(a_s)(b_r, a_r.\mathrm{arg})[\![\langle a_r\rangle \mapsto X(a_s)(b_t, a_t.\mathrm{arg})(\langle a_t.\mathrm{arg}\rangle.2)]\!] & \dot{\subseteq} & Y(a_s)(b_r, a_r) & \\
 & & & \\
X(a_s')(b_t', a_t.\mathrm{arg})[\![\langle a_t\rangle \mapsto \{(b_t', a_t)\}]\!] & \dot{\subseteq} & Y(a_s')(b_t', a_t) & \text{if } \mathit{reach}(X(a_s')(b, a_s)) \ \wedge\ b \otimes b_t = b_t' \\
X(a_s')(b_r', a_r.\mathrm{arg})[\![\langle a_r\rangle \mapsto X(a_s')(b_t', a_t.\mathrm{arg})(\langle a_t.\mathrm{arg}\rangle.2)]\!] & \dot{\subseteq} & Y(a_s')(b_r', a_r) & \text{if } \mathit{reach}(X(a_s')(b, a_s)) \ \wedge\ b \otimes b_r = b_r' \\
 & & & \\
X(a_s')(b_t', a_t.\mathrm{arg})[\![\langle a_t\rangle \mapsto \{(b_t', a_t)\}]\!] & \dot{\subseteq} & Y(a_s')(b_t', a_t) & \text{if } \mathit{reach}(X(a_s)(b, a_s')) \ \wedge\ b \otimes b_t' = b_t \\
X(a_s')(b_r', a_r.\mathrm{arg})[\![\langle a_r\rangle \mapsto \mathit{source}(X(a_s)(b_t, a_t.\mathrm{arg})(\langle a_t.\mathrm{arg}\rangle.2))]\!] & \dot{\subseteq} & Y(a_s')(b_r', a_r) & \text{if } \mathit{reach}(X(a_s)(b, a_s')) \ \wedge\ b \otimes b_r' = b_r
\end{array}$$

where $(z, \hat{\mathbf{a}}) \otimes b = (z, \hat{\mathbf{a}} \otimes b)$ and where *source* is defined in Figure 3.

Figure 5: Synchronization rules of the abstract transition function $\widehat{T}$, parameterized by $\otimes \in \widehat{CP} \times \mathit{Branch} \rightarrow \widehat{CP}$.

nizations must thus be of the form:



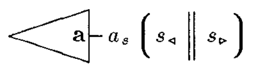
where

$$\mathit{partition}(\mathbf{a}_t) = \textsc{orig} \qquad \mathit{partition}(\mathbf{a}_r) = \textsc{orig}$$

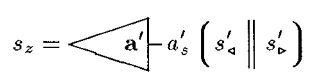
$$(a_s = \boxed{\mathbf{SL}} \ \wedge\ a_r = \boxed{\mathbf{RL}}) \ \vee\ (a_s = \boxed{\mathbf{SR}} \ \wedge\ a_r = \boxed{\mathbf{RR}})$$

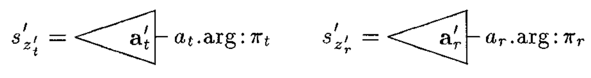
and similarly for $\boxed{\mathbf{T'}}$. This means that either the base-case or the recursive-case `transmit` in a process spawned at $\boxed{\mathbf{SL}}$ can synchronize with the `receive` at $\boxed{\mathbf{RL}}$ of the continuation of its parent process, and similarly for $\boxed{\mathbf{SR}}$/$\boxed{\mathbf{RR}}$. These are the only communications that can occur in the node code, independent of `base`, `combine`, or the other code in P.

There are other transmit/receive pairs that match, but we don't have to explicitly consider them. Consider the case where $z_t = z_r = z$. Such matches (*e.g.*, where $a_s = \boxed{\mathbf{SL}}$, $z = \triangleleft$, $\hat{\mathbf{a}}_t = \textsc{new}$, and $\hat{\mathbf{a}}_r = \textsc{orig}$) correspond to synchronizations of the form:



where





$$z_t' \neq z_r'$$

$$\mathit{partition}(\mathbf{a}'a_s'\mathbf{a}_t') = \hat{\mathbf{a}}_t \qquad \mathit{partition}(\mathbf{a}'a_s'\mathbf{a}_r') = \hat{\mathbf{a}}_r$$

But these synchronizations are covered by a transmit in row $[a_s'\ \ z_t'\ \ \hat{\mathbf{a}}_t']$ and a receive in row $[a_s'\ \ z_r'\ \ \hat{\mathbf{a}}_r']$ for some $\hat{\mathbf{a}}_t'$ and $\hat{\mathbf{a}}_r'$, and because $z_t' \neq z_r'$ this must fall into one of the two true matches already considered above. This is a rather subtle point that illustrates the benefit of the non-uniformity of the analysis—the universal quantification over control path prefixes in the proposition that was described in Section 6. The technique is that we relate pairs of processes with their point of branching. In contrast, any uniform analysis would have to consider these "false" entries as well and would thus only determine that *any* process can communicate with *any* other. Thus, the tree topology that our analysis yields would completely collapse. Therefore, we see that non-uniformity is a crucial property for a communication analysis.

9 Complexity

The height of the lattice $(\widehat{State}, \dot{\subseteq})$ is $s(n(n+v) + 2n^2p(n+v)(2p+1))$, where $s = |\mathit{SAddr}|$, $n = |\mathit{Addr}|$ (the number of expressions in P), $v = |\mathit{Var}|$, and $p = |\widehat{CP}|$. This is $O(p^2sn^3)$, and thus so is the number of iterations in the computation of the least fixpoint of $\widehat{T}$. Clearly, $\widehat{T}$ is $O(\mathit{poly}(n))$, since it is doing a local computation at each label in P, and thus the algorithm is $O(\mathit{poly}(n))$.

Our goal in this paper was to present the analysis conceptually, not to minimize the polynomial factor. However, we conjecture that $\widehat{State}$ could be reformulated, albeit with a loss of conceptual clarity, with a height of $O(p^2sn^2)$, using ideas of, *e.g.*, [Hei92, JM82]. Furthermore, for similar reasons we conjecture that there is an iterative fixpoint algorithm that is $O(p^2sn^3)$. Finally, again for similar reasons, we would expect such an algorithm to be near $O(p^2sn)$ in practice.

For our node example, $h = 2$, so we conjecture a complexity of $O(sn^3)$ worst case, and near $O(sn)$ in practice.

10 Conclusion

We have developed a powerful, and most likely practical, general-purpose communication-topology analysis for a subset of Concurrent ML with dynamic process and channel creation. The analysis uses non-uniform relational techniques that allow it to distinguish infinite communication patterns

			$\langle\boxed{T}.\text{arg}\rangle.1 \mathbin{@} \boxed{T}.\text{arg}$ $\langle\boxed{T'}.\text{arg}\rangle.1 \mathbin{@} \boxed{T'}.\text{arg}$	$\langle\boxed{RL}.\text{arg}\rangle \mathbin{@} \boxed{RL}.\text{arg}$	$\langle\boxed{RR}.\text{arg}\rangle \mathbin{@} \boxed{RR}.\text{arg}$
$\boxed{SL}$	$\triangleleft$	ORIG	$\widehat{V}_? + \{\boxed{(\text{EXTERNAL},\boxed{L})}\}$	$\{(\text{ORIG}^{\triangleleft},\boxed{L})\}$	$\{(\text{ORIG}^{\triangleleft},\boxed{R})\}$
		NEW	$\widehat{V}_? + \{(\text{ORIG}^{\triangleleft},\boxed{L}), (\text{ORIG}^{\triangleleft},\boxed{R}), (\text{NEW}^{\triangleleft},\boxed{L}), (\text{NEW}^{\triangleleft},\boxed{R})\}$	$\{(\text{NEW}^{\triangleleft},\boxed{L})\}$	$\{(\text{NEW}^{\triangleleft},\boxed{R})\}$
	$\triangleright$	ORIG	$\widehat{V}_?$	$\{\boxed{(\text{EXTERNAL},\boxed{L})}, (\text{ORIG}^{\triangleright},\boxed{L})\}$	$\{(\text{EXTERNAL},\boxed{R}), (\text{ORIG}^{\triangleright},\boxed{R})\}$
		NEW	$\widehat{V}_? + \{(\text{EXTERNAL},\boxed{R}), (\text{ORIG}^{\triangleright},\boxed{L}), (\text{ORIG}^{\triangleright},\boxed{R}), (\text{NEW}^{\triangleright},\boxed{L}), (\text{NEW}^{\triangleright},\boxed{R})\}$	$\{(\text{NEW}^{\triangleright},\boxed{L})\}$	$\{(\text{NEW}^{\triangleright},\boxed{R})\}$
$\boxed{SR}$	$\triangleleft$	ORIG	$\widehat{V}_? + \{\boxed{(\text{EXTERNAL},\boxed{R})}\}$	$\{(\text{ORIG}^{\triangleleft},\boxed{L})\}$	$\{(\text{ORIG}^{\triangleleft},\boxed{R})\}$
		NEW	$\widehat{V}_? + \{(\text{ORIG}^{\triangleleft},\boxed{L}), (\text{ORIG}^{\triangleleft},\boxed{R}), (\text{NEW}^{\triangleleft},\boxed{L}), (\text{NEW}^{\triangleleft},\boxed{R})\}$	$\{(\text{NEW}^{\triangleleft},\boxed{L})\}$	$\{(\text{NEW}^{\triangleleft},\boxed{R})\}$
	$\triangleright$	ORIG	$\widehat{V}_?$	$\{(\text{EXTERNAL},\boxed{L}), (\text{ORIG}^{\triangleright},\boxed{L})\}$	$\{\boxed{(\text{EXTERNAL},\boxed{R})}, (\text{ORIG}^{\triangleright},\boxed{R})\}$
		NEW	$\widehat{V}_? + \{(\text{ORIG}^{\triangleright},\boxed{L}), (\text{ORIG}^{\triangleright},\boxed{R}), (\text{NEW}^{\triangleright},\boxed{L}), (\text{NEW}^{\triangleright},\boxed{R})\}$	$\{(\text{NEW}^{\triangleright},\boxed{L})\}$	$\{(\text{NEW}^{\triangleright},\boxed{R})\}$
a_{other}	$\triangleleft$	ORIG	$\widehat{V}_?$	$\{(\text{ORIG}^{\triangleleft},\boxed{L})\}$	$\{(\text{ORIG}^{\triangleleft},\boxed{R})\}$
		NEW	$\widehat{V}_? + \{(\text{ORIG}^{\triangleleft},\boxed{L}), (\text{ORIG}^{\triangleleft},\boxed{R}), (\text{NEW}^{\triangleleft},\boxed{L}), (\text{NEW}^{\triangleleft},\boxed{R})\}$	$\{(\text{NEW}^{\triangleleft},\boxed{L})\}$	$\{(\text{NEW}^{\triangleleft},\boxed{R})\}$
	$\triangleright$	ORIG	$\widehat{V}_?$	$\{(\text{ORIG}^{\triangleright},\boxed{L})\}$	$\{(\text{ORIG}^{\triangleright},\boxed{R})\}$
		NEW	$\widehat{V}_? + \{(\text{ORIG}^{\triangleright},\boxed{L}), (\text{ORIG}^{\triangleright},\boxed{R}), (\text{NEW}^{\triangleright},\boxed{L}), (\text{NEW}^{\triangleright},\boxed{R})\}$	$\{(\text{NEW}^{\triangleright},\boxed{L})\}$	$\{(\text{NEW}^{\triangleright},\boxed{R})\}$

Table 1: Results of the analysis for `fold`.

where uniform analyses would fail. In contrast to sequential languages, these techniques seem to be a requirement for useful concurrent-language analyses because of the difficulty caused by dynamic processor allocation. Our work is to the best of our knowledge the first such analysis, and as such is a major conceptual step toward a new understanding of analysis problems for concurrent languages. In particular, our analysis derives information that is required for sequentialization and is useful for processor allocation, and we believe that it will prove beneficial in both uniprocessor and multiprocessor implementations of these languages.

Acknowledgments I would like to thank the Laboratoire d'Informatique of École Polytechnique for hosting me during this work. Also, thanks to Patrick Cousot, Radhia Cousot, Robert Harper, and Peter Lee for comments and suggestions.

References

[BMT92] Dave Berry, Robin Milner, and David N Turner A semantics for ML concurrency primitives In *Nineteenth Annual ACM Symposium on Principles of Programming Languages*, January 1992

[CC77] P Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Fourth Annual ACM Symposium on Principles of Programming Languages*, 1977

[CC80] P. Cousot and R. Cousot Semantic analysis of communicating sequential processes In *Proc. of 7th Int. Colloquium on Automata, Languages and Programming*, LNCS vol. 85 Springer, 1980

[CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks *Journal of Logic and Computation*, 2(4) 511–547, 1992

[CH92] Jyh-Herng Chow and Williams Ludwell Harrison Compile time analysis of parallel programs that share memory In *Nineteenth Annual ACM Symposium on Principles of Programming Languages*, January 1992.

[Col] Christopher Colby Analysis of synchronization and aliasing with abstract interpretation Unpublished

[Deu92] Alain Deutsch *Operational Models of Programming Languages and Representations of Relations on Regular Languages with Application to the Static Determination of Dynamic Aliasing Properties of Data.* PhD thesis, LIX, Ecole Polytechnique, Palaiseau, France, 1992.

[Har89] Williams Ludwell Harrison The interprocedural analysis and automatic parallelisation of scheme programs *Lisp and Symbolic Computation*, 2(3) 176–396, October 1989

[Hei92] Nevin Heintze. *Set Based Program Analysis* PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1992.

[JM82] N D. Jones and S Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures In *Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 64–74, 1982

[JW94] Suresh Jagannathan and Stephen Weeks Analyzing stores and references in a parallel symbolic language In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 294–306, 1994

[Mer91] N. Mercouroff An algorithm for analysing communicating processes In S Brookes, M Main, A. Melton, M. Mislove, and D Schmidt, editors, *Mathematical Foundations of Programming Semantics* Springer LNCS vol 598, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990

[NN94] Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology. In *Twenty-first Annual ACM Symposium on Principles of Programming Languages*, 1994.

[Rep92] John Reppy. *High-Order Concurrency.* PhD thesis, Cornell University, Ithaca, New York, June 1992