Static and Dynamic Processor Allocation for Higher-Order Concurrent Languages

Hanne Riis Nielson, Flemming Nielson

Computer Science Department,
Aarhus University, Denmark.

e-mail:{hrnielson,fnielson}@daimi.aau.dk

phone:+45.89.42.32.76 fax:+45.89.42.32.55

Abstract

Starting from the process algebra for Concurrent ML we develop two program analyses that facilitate the intelligent placement of processes on processors. Both analyses are obtained by augmenting an inference system for counting the number of channels created, the number of input and output operations performed, and the number of processes spawned by the execution of a Concurrent ML program. One analysis provides information useful for making a static decision about processor allocation; to this end it accumulates the communication cost for all processes with the same label. The other analysis provides information useful for making a dynamic decision about processor allocation; to this end it determines the maximum communication cost among processes with the same label. We prove the soundness of the inference system and the two analyses and demonstrate how to implement them; the latter amounts to transforming the syntax-directed inference problems to instances of syntax-free equation solving problems.

1 Introduction

Higher-order concurrent languages as CML [15] and FACILE [4] offer primitives for the dynamic creation of processes and channels. A distributed implementation of these languages immediately raises the problem of processor allocation. The efficiency of the implementation will depend upon how well the *network configuration* matches the *communication topology* of the program – and here it is important which processes reside on which processors. When deciding this it will be useful to know:

- Which channels will be used by the process for input and output operations and how many times will the operations be performed?

- Which channels and processes will be created by the process and how many instances will be generated?

As an example, two processes that frequently communicate with one another should be allocated on processors in the network so as to ensure a low communication overhead.

In CML and FACILE processes and channels are created dynamically and this leads naturally to a distinction between two different processor allocation schemes:

- *Static processor allocation*: At compile-time it is decided where all instances of a process will reside at run-time.
- *Dynamic processor allocation*: At run-time it is decided where the individual instances of a process will reside.

The first scheme is the simpler one and it is used in the current distributed implementation of FACILE; finer grain control over parallelism may be achieved using the second scheme [17].

What has been accomplished. In this paper we present *analyses providing information for static and dynamic processor allocation* of CML programs. We shall follow the approach of [12] and develop the analyses in two stages:

- Extract the communication behaviour of the CML program.
- Analyse the behaviour.

The two analyses only differ in the second stage.

The *first stage* follows [12] in developing a type and behaviour inference system for expressing the communication capabilities of programs in CML. This formulation takes full account of the polymorphism present in ML and an algorithm for the automatic extraction of behaviours from CML programs is developed in [13]. As was already indicated in [11] the behaviours may be regarded as terms in a process algebra (like CCS or CSP); however the process algebra of behaviours is specifically designed so as to capture those aspects of communication that are relevant for the efficient implementation of programs in CML.

The *second stage* of the two analyses are developed in detail in the present paper. To prepare for this we first develop an analysis that uses simple ideas from abstract interpretation to count for *each behaviour* the number of channels created, the number of input and output operations performed and the number of processes spawned. To provide information for static and dynamic processor allocation we then differentiate the information with respect to *labels* associated with the `fork` operations of the CML program; these labels will identify all instances of a given process and *for each label* we count the number of channels created, the number of input and output operations performed and the number of processes spawned. The central observation is now that for the *static* allocation scheme we *accumulate* the requirements of the individual instances whereas for the *dynamic* allocation scheme we take the *maximum* of the individual instance requirements.

In this paper we prove the *correctness* of the second stage of the analysis. The analyses are specified as inference systems and the correctness proof is based on a structural operational semantics for behaviours and an appropriate abstraction of the non-negative natural numbers. The correctness of the complete analysis then follows from the subject reduction result of [12] that allows us to "lift" safety (as opposed to liveness) results from the behaviours to safety results for CML programs.

We also address the *implementation* of the second stage of the analysis. Here the idea is to transform the problem as specified by the syntax-directed inference system into a syntax-free equation solving problem where standard techniques from data flow analysis can be used to obtain fast implementations. (As already mentioned the implementation of the first stage is the topic of [13].)

Comparison with other work. First we want to stress that our approach to processor allocation is that of *static program analysis* rather than, say, heuristics based on profiling as is often found in the literature on implementation of concurrent languages.

In the literature there are only few program analyses for combined functional and concurrent languages. An extension of SML with Linda communication primitives is studied in [2] and, based on the corresponding process algebra, an analysis is presented that provides useful information for the placement of processes on a finite number of processors. A functional language with communication via shared variables is studied in [8] and its communication patterns are analysed, again with the goal of producing useful information for processor (and storage) allocation. Also a couple of program analyses have been developed for concurrent languages with an imperative facet. The papers [3, 7, 14] all present reachability analyses for concurrent programs with a statically determined communication topology; only [14] shows how this restriction can be lifted to allow communication in the style of the π-calculus. Finally, [10] presents an analysis determining the number of communications on each channel connecting two processes in a CSP-like language.

As mentioned our analysis is specified in two stages. The *first stage* is formalised in [12, 13]; similar considerations were carried out by Havelund and Larsen leading to a comparable process algebra [5] but with *no* formal study of the link to CML *nor* with any algorithm for automatically extracting behaviours. The same overall idea is present in [2] but again with *no* formal study of the link between the process algebra and the programming language.

The *second stage* of the analysis extracts much more detailed information from the behaviours and this leads to a much more complex notion of correctness than in [12]. Furthermore, the analysis is parameterised on the choice of value space thereby incorporating ideas from abstract interpretation.

In summary, we believe that this paper presents the first provenly correct static analyses giving useful information for processor allocation in a higher-order language with concurrency primitives based on synchronous message passing.

2 Behaviours

Following [12] the syntax of *behaviours* (i.e. terms in the process algebra) $b \in \mathbf{Beh}$ is given by

$$b \quad ::= \quad \epsilon \mid L!t \mid L?t \mid t\ \mathtt{CHAN}_L \mid \beta \mid \mathtt{FORK}_L\ b \mid b_1; b_2 \mid b_1 + b_2 \mid \mathtt{REC}\beta.\, b$$

where $L \subseteq \mathbf{Labels}$ is a non-empty and finite set of program labels. The behaviour ϵ is associated with the pure functional computations of CML. The behaviours $L!t$ and $L?t$ are associated with sending and receiving values of type t over channels with label in L, the behaviour $t\ \mathtt{CHAN}_L$ is associated with creating a new channel with label in L and over which values of type t can be communicated, and the behaviour $\mathtt{FORK}_L\ b$ is associated with creating a new process with behaviour b and with label in L. Together these behaviours constitute the *atomic behaviours* $p \in \mathbf{ABeh}$ as may be expressed by setting:

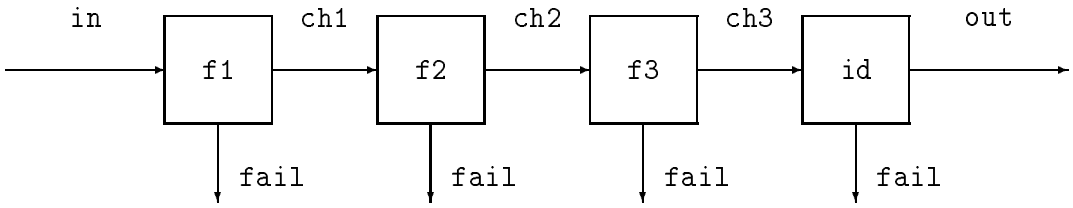
$$p \quad ::= \quad \epsilon \mid L!t \mid L?t \mid t\ \mathtt{CHAN}_L \mid \mathtt{FORK}_L\ b$$

Finally, behaviours may be composed by sequencing (as in $b_1; b_2$) and internal choice (as in $b_1 + b_2$) and we use *behaviour variables* together with an explicit `REC` construct to express recursive behaviours. The structure of the types shall be of little concern to us in this paper but for the sake of completeness we mention that the syntax of $t \in \mathbf{Typ}$ is given by

$$t \quad ::= \quad \mathtt{unit} \mid \mathtt{bool} \mid \mathtt{int} \mid \alpha \mid t_1 \rightarrow^b t_2 \mid t_1 \times t_2 \mid t\ \mathtt{list} \mid t\ \mathtt{chan}\ L \mid t\ \mathtt{com}\ b$$

where α is a meta-variable for type variables; see [12] for details.

Example 2.1 Suppose we want to construct a function `pipe` such that the call `pipe [f1,f2,f3] in out` will produce a pipe of four processes as depicted in:



Here the sequence of inputs is taken over channel `in`, the sequence of outputs is produced over channel `out` and the functions `f1`, `f2`, `f3` (and the identity function `id` defined by `fn x => x`) are applied in turn. To achieve concurrency we want separate processes for each of the functions `f1`, `f2`, `f3` (and `id`); these are interconnected using the new internal channels `ch1`, `ch2`, and `ch3`. Finally `fail` is a channel over which failure of operation may be reported.

We shall see that the following CML program will do the job:

```
let node = fn f => fn in => fn out =>
    fork_π (rec loop d =>
          sync (choose [wrap (receive in, fn x => sync (send (out, f x));
                                                loop d),
                        send(fail,())]))
in rec pipe fs => fn in => fn out =>
   if isnil fs
   then node (fn x => x) in out
   else let ch = channel ()
        in (node (hd fs) in ch; pipe (tl fs) ch out)
```

To explain this program consider first the function `node`. Here `f` is the function to be applied, `in` is the input channel and `out` is the output channel. The function $\texttt{fork}_\pi$ creates a new process labelled π that performs as described by the recursive function `loop` that takes the dummy parameter `d`. In each recursive call the function may either report failure by `send(fail,())` or it may perform one step of the processing: receive the input by means of `receive in`, take the value `x` received and transmit the modified value `f x` by means of `send(out,f x)` after which the process repeats itself by means of `loop d`. The primitive `choose` allows to perform an unspecified choice between the two communication possibilities and `wrap` allows to modify a communication by postprocessing the value received or transmitted. The `sync` primitive enforces synchronisation at the right points and we refer to [15] for a discussion of the language design issues involved in this; once we have arrived at the process algebra such considerations will be of little importance to us. Next consider the function `pipe` itself. Here `fs` is the list of functions to be applied, `in` is the input channel, and `out` is the output channel. If the list of functions is empty we connect `in` and `out` by means of a process that applies the identity function; otherwise we create a new internal channel by means of `channel ()` and then we create the process for the first function in the list and then recurse on the remainder of the list.

In the remainder of this paper we shall not be overly concerned with the syntax of CML. However it is important for us that the type inference system of [12] can be used to prove that the above program has type

$$(\alpha \to^{\beta} \alpha)\ \texttt{list} \to^{\epsilon} \alpha\ \texttt{chan}_{L_1} \to^{\epsilon} \alpha\ \texttt{chan}_{L_2} \to^{b} \texttt{unit}$$

where b is

$$\begin{aligned}&\textsc{rec}\,\beta'.(\textsc{fork}_\pi(\textsc{rec}\,\beta''.(L_1?\alpha;\epsilon;L_2!\alpha;\beta'' + L!\texttt{unit}))\\ &\quad +\alpha\ \textsc{chan}_{L_1};\textsc{fork}_\pi(\textsc{rec}\,\beta''.(L_1?\alpha;\beta;L_2!\alpha;\beta'' + L!\texttt{unit}));\beta')\end{aligned}$$

and where we assume that `fail` is a channel of type `unit chan`$_L$.

Thus the behaviour expresses directly that the `pipe` function is recursively defined and that it either spawns a single process or creates a channel, spawns a process and recurses. The spawned processes will all be recursive and they will either input over a channel in L_1, do something (as expressed by ϵ and β), output over a channel in L_2 and recurse or they will output over a "failure"-channel in L and terminate. □

$$p \Rightarrow^{p} \epsilon \qquad\qquad \epsilon \Rightarrow^{\epsilon} \surd$$

$$b \Rightarrow^{\epsilon} b \qquad\qquad \texttt{REC}\ \beta.\ b \Rightarrow^{\epsilon} b[\beta \mapsto \texttt{REC}\ \beta.\ b]$$

$$\frac{b_1 \Rightarrow^{p} b_1'}{b_1; b_2 \Rightarrow^{p} b_1'; b_2} \qquad\qquad \frac{b_1 \Rightarrow^{p} \surd}{b_1; b_2 \Rightarrow^{p} b_2}$$

$$\frac{b_1 \Rightarrow^{p} b_1'}{b_1 + b_2 \Rightarrow^{p} b_1'} \qquad\qquad \frac{b_2 \Rightarrow^{p} b_2'}{b_1 + b_2 \Rightarrow^{p} b_2'}$$

Table 1: Sequential Evolution

The *sequential* evolution of behaviours is defined in Table 1. Here the configurations of the transition system are either closed behaviours (i.e. having no free behaviour variables) or the special terminating configuration $\surd$. The transition relation takes one of the two forms

$$b \Rightarrow^{p} b' \qquad \text{and} \qquad b \Rightarrow^{p} \surd$$

where p is an atomic behaviour. The axiom $p \Rightarrow^{p} \epsilon$ allows performing the primitive behaviour p leaving the resulting behaviour ϵ; we use ϵ rather than $\surd$ to accomodate the axiom $b; \epsilon \equiv b$ of [12]. The axiom $b \Rightarrow^{\epsilon} b$ allows to perform any number of "silent" ϵ steps; this is to accomodate the axiom $\epsilon; b \equiv b$ of [12]. Less formally the idea is that *any* number of computation steps may be performed in the pure functional part of CML before or after any of the communicating steps are performed. The axiom $\epsilon \Rightarrow^{\epsilon} \surd$ expresses that the execution of the ϵ-behaviour can terminate[1]. The axiom involving REC allows to unfold the recursive construct while performing a "silent" step. The rules for sequencing are straightforward: when executing $b_1; b_2$ we are only allowed to start the execution of b_2 when b_1 has terminated. The rules for choice express an internal choice[2].

The *concurrent evolution* of behaviours is defined in Table 2. Here we associate behaviours with *process identifiers* and the transitions will take the form

$$PB \Longrightarrow^{a}_{ps} PB'$$

where PB and PB' are mappings from process identifiers to *closed* behaviours and the special symbol $\surd$. Furthermore, a is an action that takes place and ps is a list of the processes that take part in the action. The *actions* are given by

$$a ::= \epsilon \mid t\ \texttt{CHAN}_L \mid \texttt{FORK}_L\ b \mid L!t?L$$

and are closely connected to the atomic behaviours. The first two rules of Table 2 embed the pure sequential computations into the concurrent system. The next two rules incorporate channel and process creation. Note that when a new process is created we record the

[1] A more general rule would be $p \Rightarrow^{p} \surd$ for all primitive behaviours p but the effect of this can be obtained in two steps $p \Rightarrow^{p} \epsilon \Rightarrow^{\epsilon} \surd$ and since we essentially ignore ϵ-behaviours the two formulations turn out to be equivalent.

[2] An alternative would be to use the axioms $b_1 + b_2 \Rightarrow^{\epsilon} b_1$ and $b_1 + b_2 \Rightarrow^{\epsilon} b_2$ but since we always allow $b_i \Rightarrow^{\epsilon} b_i$ the two formulations turn out to be equivalent.

$$\frac{b \Rightarrow^{\epsilon} \surd}{PB[pi \mapsto b] \Longrightarrow^{\epsilon}_{pi} PB[pi \mapsto \surd]}$$

$$\frac{b \Rightarrow^{\epsilon} b'}{PB[pi \mapsto b] \Longrightarrow^{\epsilon}_{pi} PB[pi \mapsto b']}$$

$$\frac{b \Rightarrow^{t\,\mathrm{CHAN}_L} b'}{PB[pi \mapsto b] \Longrightarrow^{t\,\mathrm{CHAN}_L}_{pi} PB[pi \mapsto b']}$$

$$\frac{b \Rightarrow^{\mathrm{FORK}_L\, b_0} b'}{PB[pi_1 \mapsto b] \Longrightarrow^{\mathrm{FORK}_L\, b_0}_{pi_1, pi_2} PB[pi_1 \mapsto b'][pi_2 \mapsto b_0]}$$
$$\text{if } pi_2 \notin \mathrm{Dom}(PB) \cup \{pi_1\}$$

$$\frac{b_1 \Rightarrow^{L_1!t} b_1' \qquad b_2 \Rightarrow^{L_2?t} b_2'}{PB[pi_1 \mapsto b_1][pi_2 \mapsto b_2] \Longrightarrow^{L_1!t?L_2}_{pi_1, pi_2} PB[pi_1 \mapsto b_1'][pi_2 \mapsto b_2']}$$
$$\text{if } pi_1 \neq pi_2 \text{ and } L_1 = L_2$$

Table 2: Concurrent Evolution

process identifier of the process that created it as well as its own process identifier. Finally we have a rule that facilitates communication. Here we insist that the sets of labels that are used for the communication are equal as this is in accord with the typing system of [12]; however a more general rule would result if $L_1 = L_2$ was replaced by $L_1 \cap L_2 \neq \emptyset$ or $L_1 \subseteq L_2$. In all these rules we use the convention that PB in $PB[pi \mapsto b]$ is chosen such that the explicitly mentioned pi is not in the domain $Dom(PB)$ of PB.

3 Value Spaces

In the analyses we want to predict the number of times certain events may happen. The precision as well as the complexity of the analyses will depend upon how we count so we shall parameterise the formulation of the analyses on our notion of counting.

This amounts to abstracting the non-negative integers $\mathbf{N}$ by a complete lattice ($\mathbf{Abs}$,$\sqsubseteq$). As usual we write $\bot$ for the least element, $\top$ for the greatest element, $\bigsqcup$ and $\sqcup$ for least upper bounds and $\sqcap$ for greatest lower bounds. The abstraction is expressed by a function

$$\mathcal{R} : \mathbf{N} \to_m \mathbf{Abs}$$

that is strict (has $\mathcal{R}(0) = \bot$) and monotone (has $\mathcal{R}(n_1) \sqsubseteq \mathcal{R}(n_2)$ whenever $n_1 \leq n_2$); hence the ordering on the natural numbers is reflected in the abstract values. Three elements of $\mathbf{Abs}$ are of particular interest and we shall introduce special syntax for them:

$$\mathrm{o} = \mathcal{R}(0) = \bot$$
$$\mathrm{i} = \mathcal{R}(1)$$

$\mathtt{M} = \top$

We cannot expect our notion of counting to be precisely reflected by **Abs**; indeed it is likely that we shall allow to identify for example $\mathcal{R}(2)$ and $\mathcal{R}(3)$ and perhaps even $\mathcal{R}(1)$ and $\mathcal{R}(2)$. However, we shall ensure throughout that no identifications involve $\mathcal{R}(0)$ by demanding that

$$\mathcal{R}^{-1}(\mathtt{O}) = \{0\}$$

so that O really represents "did not happen".

We shall be interested in two binary operations on the non-negative integers. One is the operation of *maximum*: $\max\{n_1, n_2\}$ is the larger of n_1 and n_2. In **Abs** we shall use the binary least upper bound operation to express the maximum operation. Indeed $\mathcal{R}(\max\{n_1, n_2\}) = \mathcal{R}(n_1) \sqcup \mathcal{R}(n_2)$ holds by monotonicity of $\mathcal{R}$ as do the laws $n_1 \sqsubseteq n_1 \sqcup n_2$, $n_2 \sqsubseteq n_1 \sqcup n_2$ and $n \sqcup n = n$. As a consequence $n_1 \sqcup n_2 = \mathtt{O}$ iff both n_1 and n_2 equal O.

The other operation is *addition*: $n_1 + n_2$ is the sum of n_1 and n_2. In **Abs** we shall have to define a function $\oplus$ and demand that

(**Abs**, $\oplus$, O) is an Abelian *monoid* with $\oplus$ monotone

This ensures that we have the associative law $n_1 \oplus (n_2 \oplus n_3) = (n_1 \oplus n_2) \oplus n_3$, the absorption laws $n \oplus \mathtt{O} = \mathtt{O} \oplus n = n$, the commutative law $n_1 \oplus n_2 = n_2 \oplus n_1$ and by monotonicity we have also the laws $n_1 \sqsubseteq n_1 \oplus n_2$ and $n_2 \sqsubseteq n_1 \oplus n_2$. As a consequence $n_1 \oplus n_2 = \mathtt{O}$ iff both n_1 and n_2 equal O. To ensure that $\oplus$ models addition on the integers we impose the condition

$$\forall n_1, n_2.\ \mathcal{R}(n_1 + n_2) \sqsubseteq \mathcal{R}(n_1) \oplus \mathcal{R}(n_2)$$

that is common in abstract interpretation.

Definition 3.1 A *value space* is a structure (**Abs**,$\sqsubseteq$,O,I,M,$\oplus$,$\mathcal{R}$) as detailed above. It is an *atomic value space* if I is an atom (that is $\mathtt{O} \sqsubseteq n \sqsubseteq \mathtt{I}$ implies that $\mathtt{O} = n$ or $\mathtt{I} = n$).

Example 3.2 One possibility is to use $\mathbf{Abs}_\infty = \mathbf{N} \cup \{\infty\}$ and define $\sqsubseteq$ as the extension of $\leq$ by $n \sqsubseteq \infty$ for all n. The abstraction function $\mathcal{R}$ can be taken as the injection function. We then have

$$n_1 \sqcup n_2 = \begin{cases} \max\{n_1, n_2\} & \text{if } n_1, n_2 \in \mathbf{N} \\ \infty & \text{otherwise} \end{cases}$$

For the operation $\oplus$ we take

$$n_1 \oplus n_2 = \begin{cases} n_1 + n_2 & \text{if } n_1, n_2 \in \mathbf{N} \\ \infty & \text{otherwise} \end{cases}$$

Clearly we have $\mathtt{O} = 0$, $\mathtt{I} = 1$ and $\mathtt{M} = \infty$. This defines an atomic value space. □

Example 3.3 Another possibility is to use $\mathbf{A3} = \{\mathtt{O}, \mathtt{I}, \mathtt{M}\}$ and define $\sqsubseteq$ by $\mathtt{O} \sqsubseteq \mathtt{I} \sqsubseteq \mathtt{M}$. The abstraction function $\mathcal{R}$ will then map 0 to O, 1 to I and all other numbers to M. The operations $\sqcup$ and $\oplus$ can then be given by the following tables:

$\sqcup$	O	I	M
O	O	I	M
I	I	I	M
M	M	M	M

$\oplus$	O	I	M
O	O	I	M
I	I	M	M
M	M	M	M

This defines an atomic value space. □

For two value spaces $(\mathbf{Abs}', \sqsubseteq', \mathtt{O}', \mathtt{I}', \mathtt{M}', \oplus', \mathcal{R}')$ and $(\mathbf{Abs}'', \sqsubseteq'', \mathtt{O}'', \mathtt{I}'', \mathtt{M}'', \oplus'', \mathcal{R}'')$ we may construct their *cartesian product* $(\mathbf{Abs}, \sqsubseteq, \mathtt{O}, \mathtt{I}, \mathtt{M}, \oplus, \mathcal{R})$ by setting $\mathbf{Abs} = \mathbf{Abs}' \times \mathbf{Abs}''$ and by defining $\sqsubseteq$, O, I, M, $\oplus$ and $\mathcal{R}$ componentwise. This defines a value space but it is not atomic even if $\mathbf{Abs}'$ and $\mathbf{Abs}''$ both are. As a consequence $\mathtt{I} = (\mathtt{I}', \mathtt{I}'')$ will be of no concern to us; instead we use $(\mathtt{O}', \mathtt{I}'')$ and $(\mathtt{I}', \mathtt{O}'')$ as appropriate.

For a value space $(\mathbf{Abs}', \sqsubseteq', \mathtt{O}', \mathtt{I}', \mathtt{M}', \oplus', \mathcal{R}')$ and a non-empty set E of *events* we may construct the *indexed value space* (or function space) $(\mathbf{Abs}, \sqsubseteq, \mathtt{O}, \mathtt{I}, \mathtt{M}, \oplus, \mathcal{R})$ by setting $\mathbf{Abs} = E \to \mathbf{Abs}'$ (the set of total functions from E to $\mathbf{Abs}'$) and by defining $\sqsubseteq$, O, I, M, $\oplus$ and $\mathcal{R}$ componentwise. This defines a value space that is only atomic when $\mathbf{Abs}'$ is and provided E is a singleton set. As a consequence $\mathtt{I} = \lambda e.\mathtt{I}'$ will be of no concern to us.

For indexed value spaces we may represent

$$(f \in E \to \mathbf{Abs}) \qquad \text{by} \qquad (rep(f) \in E \hookrightarrow \mathbf{Abs} \backslash \{\mathtt{O}\})$$

where $E \hookrightarrow \mathbf{Abs}\backslash\{\mathtt{O}\}$ denotes the set of partial functions from E to $\mathbf{Abs}\backslash\{\mathtt{O}\}$; here $rep(f)$ maps e to n iff $f(e) = n$ and $n \neq \mathtt{O}$. This involves no loss of precision because there is a bijective correspondance between the two representations. Furthermore there is never a need to decrease the domains of functions involved, i.e. $Dom(rep(f_1 \oplus f_2))$ and $Dom(rep(f_1 \sqcup f_2))$ both equal $Dom(rep(f_1)) \cup Dom(rep(f_2))$ because neither $\oplus$ nor $\sqcup$ can yield O unless both operands are O.

In practice we want to restrict E to be a finite set in order to obtain finite representations. Actually we shall allow the analyses to be a bit informal about this: effectively by pretending that E might be infinite but that the indexed value spaces operates with functions $f \in E \to_f \mathbf{Abs}$ that are O on all but a finite number of arguments. Here we still have a bijective correspondence between the f's and the $rep(f)$'s having a finite domain; the only snag is that the value space then has no greatest element but that for each finite subset of E one has to be content with having a greatest element for functions that are O outside that finite set.

4 Counting the Behaviours

For a given behaviour b and value space $\mathbf{Abs}$ we may ask the following four questions:

- How many times are channels labelled by L created?

$$benv \vdash \epsilon : [\,]$$

$$benv \vdash L!t : [L \mapsto (\mathtt{O},\mathtt{O},\mathtt{I},\mathtt{O})] \qquad benv \vdash L?t : [L \mapsto (\mathtt{O},\mathtt{I},\mathtt{O},\mathtt{O})]$$

$$benv \vdash t\ \mathtt{CHAN}_L : [L \mapsto (\mathtt{I},\mathtt{O},\mathtt{O},\mathtt{O})] \qquad \frac{benv \vdash b : A}{benv \vdash \mathtt{FORK}_L\ b : [L \mapsto (\mathtt{O},\mathtt{O},\mathtt{O},\mathtt{I})] \oplus A}$$

$$\frac{benv \vdash b_1 : A_1 \quad benv \vdash b_2 : A_2}{benv \vdash b_1; b_2 : A_1 \oplus A_2} \qquad \frac{benv \vdash b_1 : A_1 \quad benv \vdash b_2 : A_2}{benv \vdash b_1 + b_2 : A_1 \sqcup A_2}$$

$$\frac{benv[\beta \mapsto A] \vdash b : A}{benv \vdash \mathtt{REC}\ \beta.\ b : A} \qquad benv \vdash \beta : A \quad \text{if } benv(\beta) = A$$

Table 3: Analysis of behaviours

- How many times do channels labelled by L participate in input?
- How many times do channels labelled by L participate in output?
- How many times are processes labelled by L generated?

To answer these questions we define an inference system with formulae

$$benv \vdash b : A$$

where $\mathbf{LabSet} = \mathcal{P}_f(\mathbf{Labels})$ is the set of finite and non-empty subsets of **Labels** and

$$A \in \mathbf{LabSet} \to_f \mathbf{Abs}$$

records the required information.

In this section we shall define the inference system for answering all these questions simultaneously. Hence we let **Abs** be the four-fold cartesian product $\mathbf{Ab}^4$ of an atomic value space **Ab**; we shall leave the formulation parameterised on the choice of **Ab** but a useful candidate is the three-element value space **A3** of Example 3.3 and we shall use this in the examples.

The idea is that $A(L) = (n_c, c_i, n_o, n_f)$ means that channels labelled by L are created at most n_c times, that channels labelled by L participate in at most n_i input operations, that channels labelled by L participate in at most n_o output operations, and that processes labelled by L are generated at most n_f times. The behaviour environment *benv* then associates each behaviour variable with an element of $\mathbf{LabSet} \to_f \mathbf{Abs}$.

The analysis is defined in Table 3. We use $[\,]$ as a shorthand for $\lambda L.(\mathtt{O},\mathtt{O},\mathtt{O},\mathtt{O})$ and $[L \mapsto \vec{n}]$ as a shorthand for $\lambda L'.\left\{ \begin{array}{ll} (\mathtt{O},\mathtt{O},\mathtt{O},\mathtt{O}) & \text{if } L' \neq L \\ \vec{n} & \text{if } L' = L \end{array} \right\}$. Note that $\mathtt{I}$ denotes the designated "one"-element in each copy of $\mathbf{Abs}_0$ since it is the atoms $(\mathtt{I},\mathtt{O},\mathtt{O},\mathtt{O})$, $(\mathtt{O},\mathtt{I},\mathtt{O},\mathtt{O})$, $(\mathtt{O},\mathtt{O},\mathtt{I},\mathtt{O})$, and $(\mathtt{O},\mathtt{O},\mathtt{O},\mathtt{I})$ that are useful for increasing the count. In the rule for $\mathtt{FORK}_L$ we are deliberately incorporating the effects of the forked process; to avoid doing so simply

remove the "$\oplus A$" component. The rules for sequencing, choice, and behaviour variables are straightforward given the developments of the previous section.

Note that the rule for recursion expresses a fixed point property and so allows some slackness; it would be inelegant to specify a least (or greatest) fixed point property whereas a post-fixed point[3] could easily be accomodated by incorporating a notion of subsumption into the rule. We decided not to incorporate a general subsumption rule and to aim for specifying as unique results as the rule for recursion allows.

Example 4.1 For the `pipe` function of Example 2.1 the analysis will give the following information:

L_1: M channels created
M inputs performed
L_2: M outputs performed
L: M outputs performed
π: M processes created

Thus the program will create many channels in L_1 and many processes labelled π and it will communicate over the channels of L_1, L_2 and L many times. While this is evidently correct it also seems pretty uninformative; yet we shall see that this simple analysis suffices for developing more informative analyses for static and dynamic processor allocation. □

Before considering the correctness of the inference system we present a few observations about its properties. The concept of free behaviour variables of a behaviour is standard; we shall need to modify this concept and so define the set $EV(b)$ of *exposed behaviour variables* of b:

$$EV(\epsilon) = EV(L!t) = EV(L?t) = EV(t\ \textsc{chan}_L) = \emptyset$$
$$EV(\textsc{fork}_L\ b) = EV(b)$$
$$EV(b_1; b_2) = EV(b_1 + b_2) = EV(b_1) \cup EV(b_2)$$
$$EV(\textsc{rec}\beta.b) = EV(b)\backslash\{\beta\}$$
$$EV(\beta) = \{\beta\}$$

Thus the difference between free and exposed variables is that the latter do not include behaviour variables embedded in type components. This suffices for stating

Fact 4.2 Suppose $benv \vdash b : A$; if $\beta \in EV(b)$ then $A \sqsupseteq benv(\beta)$ and otherwise $benv[\beta \mapsto A_\beta] \vdash b : A$ holds for all A_β. □

For the next result we need to recall the Egli-Milner ordering:

$$X \sqsubseteq_{EM} Y \quad \text{iff} \quad (\forall x \in X.\ \exists y \in Y.\ x \sqsubseteq y) \wedge (\forall y \in Y.\ \exists x \in X.\ x \sqsubseteq y)$$

Also we shall say that a behaviour environment *benv suffices* for b when all exposed variables of b are in the domain of *benv*. We then have

[3] We take a post-fixed point of a function f to be an argument n such that $f(n) \sqsubseteq n$.

Lemma 4.3 For all b and $benv$ that suffice for b the set $\{A \mid benv \vdash b : A\}$ is non-empty and has a least and a greatest element; furthermore the set depends monotonically on $benv$ in the sense that $\{A \mid benv_1 \vdash b : A\} \sqsubseteq_{EM} \{A \mid benv_2 \vdash b : A\}$ whenever $benv_1 \sqsubseteq benv_2$ and both $benv_1$ and $benv_2$ suffice for b. □

To express the correctness of the analysis we need a few definitions. Given a list X of actions define

$$\text{COUNT}(X) = \lambda L.(CC(X,L), CI(X,L), CO(X,L), CF(X,L))$$

where

$CC(X,L)$: the number of elements of the form t $\texttt{CHAN}_L$ in X,

$CI(X,L)$: the number of elements of the form $L'!t?L$ in X,

$CO(X,L)$: the number of elements of the form $L!t?L'$ in X, and

$CF(X,L)$: the number of elements of the form $\texttt{FORK}_L\ b$ in X.

Soundness of the analysis is then established by:

Theorem 4.4 If $\emptyset \vdash b : A$ and

$$[pi_0 \mapsto b] \Longrightarrow_{ps_1}^{a_1} \ldots \Longrightarrow_{ps_k}^{a_k} PB$$

then we have

$$\mathcal{R}^*(\text{COUNT}[a_1, \cdots, a_k]) \sqsubseteq A.$$

where $\mathcal{R}^*(C)(L) = (\mathcal{R}(c), \mathcal{R}(i), \mathcal{R}(o), \mathcal{R}(f))$ whenever $C(L) = (c,i,o,f)$. □

To prove this result we need the following lemma expressing the sequential soundness of the analysis:

Lemma 4.5 If $\emptyset \vdash b : A$ and $b \Rightarrow^p b'$ then there exists A_0 and A' such that $\emptyset \vdash p : A_0$, $\emptyset \vdash b' : A'$ and $A_0 \oplus A' \sqsubseteq A$. □

Here we have extended the predicate of Table 3 to configurations by taking

$$\emptyset \vdash \surd : [\,]$$

To prove the concurrent soundness of the analysis we define

$$\vdash PB : A$$

to mean that $PB = [pi_1 \mapsto b_1, \cdots, pi_j \mapsto b_j]$, $\emptyset \vdash b_1 : A_1, \cdots, \emptyset \vdash b_j : A_j$ and $A = A_1 \oplus \cdots \oplus A_j$. We then have the following proposition from which Theorem 4.4 immediately follows:

Proposition 4.6 If $\vdash PB : A$ and

$$PB \Longrightarrow^{a_1}_{ps_1} \ldots \Longrightarrow^{a_k}_{ps_k} PB'$$

then there exists A' such that $\vdash PB' : A'$ and

$$\mathcal{R}^*(\mathrm{COUNT}[a_1, \cdots, a_k]) \oplus A' \sqsubseteq A. \qquad \Box$$

Variations on the inference system presented here are easily constructed. The entire development is parameterised on the choice of **Ab**: using $\mathbf{Abs}_\infty$ of Example 3.2 will give extremely precise answers compared with using **A3** of Example 3.3. It is also immediate to change the form of the definition of **Abs**: taking $\mathbf{Abs} = \mathbf{Ab}$ we have the right setting for answering each question individually rather than simultaneously. These variations hardly change the development at all because our analysis always succeeds; in particular we do not risk that failure of one component inflicts failure upon another component.

Another variation is to replace $A : \mathbf{LabSet} \to_f \mathbf{Abs}$ with $A' : \mathbf{Labels} \to_f \mathbf{Abs}$ that more directly gives the desired information for each label. One can always obtain information in the form of A' from information in the form of A (simply use the formula $A'(l) = \bigsqcup \{A(L) \mid l \in L\}$) but in general not vice versa. However, when the behaviours are as constructed in [12] we expect that each label occurs in at most one label set, i.e. the sets of $Dom(rep(A))$ are mutually disjoint, and then the difference between the two approaches is only minor. Either way the modifications to the inference system of Table 3 are straightforward.

Replacing $\mathbf{LabSet} \to_f \mathbf{Abs}$ by $\mathbf{Abs}^2_\infty = (\mathbf{N} \cup \{\infty\})^2$ and only counting the number of channels created and the number of processes generated is also straightforward and essentially gives the analysis for detecting multiplexing and multitasking that was developed in [12]. The major difference is that the analysis of [12] only operates over $\mathbf{N}^2$ and so has to fail if ∞ was to be produced; unlike the present approach this means that failure in one component may inflict failure upon another.

5 Implementation

It is well-known that compositional specifications of program analyses (whether as abstract interpretations or annotated type systems) are not the most efficient way of obtaining the actual solutions. We therefore demonstrate how the inference problem may be transformed to an equation solving problem that is independent of the syntax of our process algebra and where standard algorithmic techniques may be applied. This approach also carries over to the inference systems for processor allocation developed subsequently.

The first step is to generate the set of equations. To show that this does not affect the set of solutions we shall be careful to avoid undesirable "cross-over" between equations generated from disjoint syntactic components of the behaviour. One possible cause for such "cross-over" is that behaviour variables may be bound in more than one `REC`; one classical solution to this is to require that the overall behaviour be alpha-renamed such that this does not occur; the solution we adopt avoids this requirement by suitable modification

$$\mathcal{E}[\![B : \pi : \epsilon]\!] = \{\langle\pi\rangle = [\,]\,\}$$

$$\mathcal{E}[\![B : \pi : L!t]\!] = \{\langle\pi\rangle = [L \mapsto (\mathtt{O},\mathtt{O},\mathtt{I},\mathtt{O})]\,\}$$

$$\mathcal{E}[\![B : \pi : L?t]\!] = \{\langle\pi\rangle = [L \mapsto (\mathtt{O},\mathtt{I},\mathtt{O},\mathtt{O})]\,\}$$

$$\mathcal{E}[\![B : \pi : t\ \mathtt{CHAN}_L]\!] = \{\langle\pi\rangle = [L \mapsto (\mathtt{I},\mathtt{O},\mathtt{O},\mathtt{O})]\,\}$$

$$\mathcal{E}[\![B : \pi : \mathtt{FORK}_L\ b]\!] = \{\langle\pi\rangle = [L \mapsto (\mathtt{O},\mathtt{O},\mathtt{O},\mathtt{I})] \oplus \langle\pi 1\rangle\,\} \ \cup\ \mathcal{E}[\![B : \pi 1 : b]\!]$$

$$\mathcal{E}[\![B : \pi : b_1; b_2]\!] = \{\langle\pi\rangle = \langle\pi 1\rangle \oplus \langle\pi 2\rangle\,\} \ \cup\ \mathcal{E}[\![B : \pi 1 : b_1]\!] \ \cup\ \mathcal{E}[\![B : \pi 2 : b_2]\!]$$

$$\mathcal{E}[\![B : \pi : b_1 + b_2]\!] = \{\langle\pi\rangle = \langle\pi 1\rangle \sqcup \langle\pi 2\rangle\,\} \ \cup\ \mathcal{E}[\![B : \pi 1 : b_1]\!] \ \cup\ \mathcal{E}[\![B : \pi 2 : b_2]\!]$$

$$\mathcal{E}[\![B : \pi : \beta]\!] = \{\langle\pi\rangle = \langle\beta\rangle\,\}$$

$$\mathcal{E}[\![B : \pi : \mathtt{REC}\ \beta.\ b]\!] = \mathrm{CLOSE}^{\pi}_{\beta}(\ \{\langle\pi\rangle = \langle\pi 1\rangle,\ \langle\pi\rangle = \langle\beta\rangle\,\} \ \cup\ \mathcal{E}[\![B : \pi 1 : b]\!]\ )$$

Table 4: Constructing the equation system

of the equation system. Another possible cause for "cross-over" is that disjoint syntactic components of the overall behaviour may nonetheless have components that syntactically appear the same; we avoid this problem by the standard use of tree-addresses (denoted π).

The function $\mathcal{E}$ for generating the equations for the overall behaviour B achieves this by the call $\mathcal{E}[\![B : \varepsilon : B]\!]$ where ε denotes the empty tree-address. In general $B : \pi : b$ indicates that the subtree of B rooted at π is of the form b and the result of $\mathcal{E}[\![B : \pi : b]\!]$ is the set of equations produced for b. The formal definition is given in Table 4.

The key idea is that $\mathcal{E}[\![B : \pi : b]\!]$ operates with *flow variables* of the form $\langle\pi'\rangle$ and $\langle\beta'\rangle$. We shall maintain the invariant that all π' occurring in $\mathcal{E}[\![B : \pi : b]\!]$ are (possibly empty) prolongations of π and that all β' occurring in $\mathcal{E}[\![B : \pi : b]\!]$ are exposed in b. To maintain this invariant in the case of recursion we define

$$\mathrm{CLOSE}^{\pi}_{\beta}(\mathbf{E}) = \{\ (L[\langle\pi\rangle/\langle\beta\rangle] = R[\langle\pi\rangle/\langle\beta\rangle]) \quad | \quad (L = R)\ \in\ \mathbf{E}\,\}$$

(although it would actually suffice to apply the substitution $[\langle\pi\rangle/\langle\beta\rangle]$ on the righthand sides of equations and it would be correct to remove the trivial equation produced).

Terms of the equations are formal terms over the flow variables (that range over the complete lattice $E \to \mathbf{Abs}$), the operations $\oplus$ and $\sqcup$ and the constants (that are elements of the complete lattice $E \to \mathbf{Abs}$). Thus all terms are monotonic in their free flow variables. A *solution* to a set $\mathbf{E}$ of equations is a partial function σ from flow variables to $E \to \mathbf{Abs}$ such that all flow variables in $\mathbf{E}$ are in the domain of σ and such that all equations $(L = R)$ of $\mathbf{E}$ have $\sigma(L) = \sigma(R)$ where σ is extended to formal terms in the obvious way. We write $\sigma \models \mathbf{E}$ whenever this is the case.

To express the relationship between the equations and the inference system we shall introduce some notation. When F is a finite set of behaviour variables we write $benv\lceil F$

for the total function with domain F that maps $\beta \in F$ to $benv(\beta)$. Similarly we shall write $\sigma\lceil F$ for the total function with domain F that maps $\beta \in F$ to $\sigma(\langle\beta\rangle)$. (We shall take care to use these notations only when we can ensure that the the resulting functions are indeed total.) Correctness of the equations then amounts to

Theorem 5.1 The set $\{\, (benv\lceil EV(b), A) \mid benv \vdash b : A \,\}$ is equal to $\{\, (\sigma\lceil EV(b), \sigma(\langle\pi\rangle)) \mid \sigma \models \mathcal{E}[\![B : \pi : b]\!] \,\}$. □

Corollary 5.2 $[\,] \vdash b : A$ iff $\exists\sigma.\ \sigma \models \mathcal{E}[\![b : \varepsilon : b]\!] \wedge \sigma(\langle\varepsilon\rangle) = A$. □

Corollary 5.3 The least (or greatest) A such that $[\,] \vdash b : A$ is of the form $\sigma(\langle\varepsilon\rangle)$ for the least (or greatest) σ such that $\sigma \models \mathcal{E}[\![b : \varepsilon : b]\!]$. □

We have now transformed our inference problem to a form where the standard algorithmic techniques can be exploited. These include simplifications of the equation system, partitioning the equation system into strongly connected components processed in (reverse) topological order, widening to ensure convergence when **Abs** does not have finite height etc.; a good overview of useful techniques may be found in [1, 6, 9, 16]. Also the flow variables may be decomposed to families of flow variables over simpler value spaces using the isomorphisms[4]

$$\begin{array}{l}
\{1\} \to \mathbf{Abs}' \cong \mathbf{Abs}' \\
(E_1 \uplus E_2) \to \mathbf{Abs}' \cong (E_1 \to \mathbf{Abs}') \times (E_2 \to \mathbf{Abs}') \\
E \to (\mathbf{Abs}' \times \mathbf{Abs}'') \cong (E \to \mathbf{Abs}') \times (E \to \mathbf{Abs}'')
\end{array}$$

where $(E_1 \uplus E_2)$ denotes $E_1 \cup E_2$ subject to E_1 and E_2 being disjoint.

A final point worth mentioning is that we have generated a system of equations (i.e. $L = R$) rather than a system of inequations (i.e. $L \sqsupseteq R$). Given that there is a binary least upper bound operator $\sqcup$ associated with the partial order $\sqsupseteq$ there is hardly any difference between the two formulations if the expressions L and R are unconstrained in format: just model $L = R$ as $L \sqsupseteq R$ and $R \sqsupseteq L$ and model $L \sqsupseteq R$ as $L = L \sqcup R$. In our case L is constrained to be a flow variable and here the equation system is the more expressive one. Although we have only been generating equations it is interesting to point out that we would have been generating inequations if our inference system included a subsumption rule for the flow information: i.e. effectively allowing to replace any A by A' provided that $A' \sqsupseteq A$.

To further clarify the relationship between equations and inequations consider a set $\mathbf{E}$ of inequations and the following operations on it. By $\mathbf{E}^{\sqcup}$ we denote the inequation system where all inequations $L \sqsupseteq R_1, \cdots, L \sqsupseteq R_n$ with the same lefthandside are "coalesced" into the single inequation $L \sqsupseteq R_1 \sqcup \cdots \sqcup R_n$. (Extensions of this procedure would remove any R_i being equal to L and would remove $R \sqsupseteq \bot$ altogether.) Further let $\mathbf{E}^{=}$ (and similarly $\mathbf{E}^{\sqcup=}$) denote the system $\mathbf{E}$ (and similarly $\mathbf{E}^{\sqcup}$) where all inequations $(L \sqsupseteq R)$ have been transformed into equations $(L = R)$. Writing $\mathcal{S}(\mathbf{E}')$ for the set of solutions to the system $\mathbf{E}'$ and $\mu(\mathbf{E}')$ for the least solution we have

[4] An isomorphism θ from $(\mathbf{Abs}', \sqsubseteq', \mathrm{O}', \mathrm{I}', \mathrm{M}', \oplus', \mathcal{R}')$ to $(\mathbf{Abs}'', \sqsubseteq'', \mathrm{O}'', \mathrm{I}'', \mathrm{M}'', \oplus'', \mathcal{R}'')$ is a bijective function θ from $\mathbf{Abs}'$ to $\mathbf{Abs}''$ such that θ and θ^{-1} are monotone and $\mathrm{O}'' = \theta(\mathrm{O}')$, $\mathrm{I}'' = \theta(\mathrm{I}')$, $\mathrm{M}'' = \theta(\mathrm{M}')$, $n_1 \oplus'' n_2 = \theta((\theta^{-1} n_1) \oplus' (\theta^{-1} n_2))$ and $\mathcal{R}'' = \theta \circ \mathcal{R}'$.

$$\mathcal{S}(\mathbf{E}^{=}) \subseteq \mathcal{S}(\mathbf{E}^{\sqcup=}) \subseteq \mathcal{S}(\mathbf{E}^{\sqcup}) = \mathcal{S}(\mathbf{E})$$
$$\mu(\mathbf{E}^{\sqcup=}) = \mu(\mathbf{E}^{\sqcup}) = \mu(\mathbf{E}) \sqsubseteq \mu(\mathbf{E}^{=})$$

where the latter inequality may be strict (e.g. for $\mathbf{E} = \{\langle\beta_1\rangle \sqsupseteq \langle\beta_2\rangle,\ \langle\beta_1\rangle \sqsupseteq \mathtt{I}\}$). So when least fixed points are sought of "coalesced" systems there is no difference between equational and inequational form.

6 Static Processor Allocation

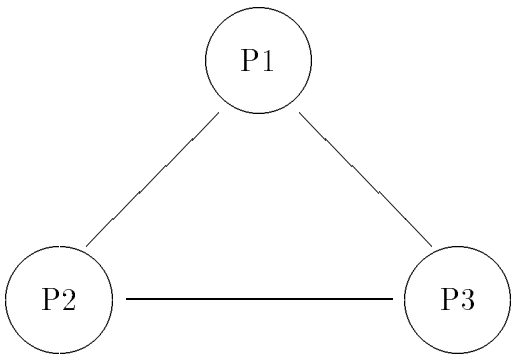
The idea behind the static processor allocation is that all processes with the *same* label will be placed on the *same* processor and we would therefore like to know what requirements this puts on the processor. To obtain such information we shall extend the simple counting analysis of Section 4 to associate information with the *process labels* mentioned in a given behaviour b. For each process label L_a we therefore ask the four questions of Section 4 accumulating the *total* information for all processes with label L_a: how many times are channels labelled by L created, how many times do channels labelled by L participate in input, how many times do channels labelled by L participate in output, and how many times are processes labelled by L generated?

Example 6.1 Let us return to the `pipe` function of Example 2.1 and suppose that we want to perform *static processor allocation*. This means that *all* instances of the processes labelled π will reside on the *same* processor. The analysis should therefore estimate the *total* requirements of these processes as follows:

main program:	L_1:	$\mathtt{M}$ channels created
	π:	$\mathtt{M}$ processes created
processes π:	L_1:	$\mathtt{M}$ inputs performed
	L_2:	$\mathtt{M}$ outputs performed
	L:	$\mathtt{M}$ outputs performed

Note that even though each process labelled by π can only communicate once over L we can generate many such processes and their combined behaviour is to communicate many times over L. It follows from this analysis that the main program does not in itself communicate over L_2 or L and that the processes do not by themselves spawn new processes.

Now suppose we have a network of processors that may be explained graphically as follows:



$$benv \vdash \epsilon : [\,] \;\&\; [\,]$$

$$benv \vdash L!t : [L \mapsto (\mathtt{O},\mathtt{O},\mathtt{I},\mathtt{O})] \;\&\; [\,] \qquad benv \vdash L?t : [L \mapsto (\mathtt{O},\mathtt{I},\mathtt{O},\mathtt{O})] \;\&\; [\,]$$

$$benv \vdash t\ \textsc{chan}_L : [L \mapsto (\mathtt{I},\mathtt{O},\mathtt{O},\mathtt{O})] \;\&\; [\,] \qquad \frac{benv \vdash b : A \;\&\; P}{benv \vdash \textsc{fork}_L\ b : [L \mapsto (\mathtt{O},\mathtt{O},\mathtt{O},\mathtt{I})] \;\&\; ([L \mapsto A] \oplus P)}$$

$$\frac{benv \vdash b_1 : A_1 \;\&\; P_1 \quad benv \vdash b_2 : A_2 \;\&\; P_2}{benv \vdash b_1;b_2 : A_1 \oplus A_2 \;\&\; P_1 \oplus P_2} \qquad \frac{benv \vdash b_1 : A_1 \;\&\; P_1 \quad benv \vdash b_2 : A_2 \;\&\; P_2}{benv \vdash b_1 + b_2 : A_1 \sqcup A_2 \;\&\; P_1 \sqcup P_2}$$

$$\frac{benv[\beta \mapsto A \;\&\; P] \vdash b : A \;\&\; P}{benv \vdash \textsc{rec}\ \beta.\, b : A \;\&\; P} \qquad benv \vdash \beta : A \;\&\; P \qquad \text{if } benv(\beta) = A \;\&\; P$$

Table 5: Analysis for static process allocation

One way to place our processes is to place the main program on **P1** and all the processes labelled π on **P2**. This requires support for multitasking on **P2** and for multiplexing (over L_1) on **P1** and **P2**. □

The analysis is obtained by modifying the inference system of Section 4 to have formulae

$$benv \vdash b : A \;\&\; P$$

where $A \in \mathbf{LabSet} \to_f \mathbf{Abs}$ as before and

$$P : \mathbf{LabSet} \to_f (\mathbf{LabSet} \to_f \mathbf{Abs})$$

The idea is that if some process is labelled L_a then $P(L_a)$ describes the *total* requirements of all processes labelled by L_a. The behaviour environment *benv* is an extension of that of Section 4 in that it associates pairs $A \;\&\; P$ with the behaviour variables. Note that in the rule for $\textsc{fork}_L$ we have removed the "$\oplus A$" component from the local effect; instead it is incorporated in the global effect for L.

To express the correctness of the analysis we need to keep track of the relationship between the process identifiers and the associated labels. So let *penv* be a mapping from process identifiers to elements L_a of **LabSet**. We shall say that *penv* respects the derivation sequence $PB \Longrightarrow_{ps_1}^{a_1} \ldots \Longrightarrow_{ps_k}^{a_k} PB'$ if whenever (a_i, ps_i) have the form $(\textsc{fork}_L\ b, (pi_1, pi_2))$ then $penv(pi_2) = L$; this ensures that the newly created process (pi_2) indeed has a label (in L) as reported by the semantics.

We can now redefine the function COUNT of Section 4. Given a list $\mathcal{X}$ of pairs of actions and lists of process identifiers define

$$\mathrm{COUNT}^{penv}(\mathcal{X}) = \lambda L_a.\lambda L.(CC_{L_a}(\mathcal{X},L), CI_{L_a}(\mathcal{X},L), CO_{L_a}(\mathcal{X},L), CF_{L_a}(\mathcal{X},L))$$

where

$CC_{L_a}(\mathcal{X}, L)$: the number of elements of the form $(t\ \textsc{chan}_L, pi)$ in $\mathcal{X}$ where $penv(pi) = L_a$,

$CI_{L_a}(\mathcal{X}, L)$: the number of elements of the form $(L'!t?L, (pi', pi))$ in $\mathcal{X}$, where $penv(pi) = L_a$,

$CO_{L_a}(\mathcal{X}, L)$: the number of elements of the form $(L!t?L', (pi, pi'))$ in $\mathcal{X}$, where $penv(pi) = L_a$, and

$CF_{L_a}(\mathcal{X}, L)$: the number of elements of the form $(\textsc{fork}_L b, (pi, pi'))$ in $\mathcal{X}$ where $penv(pi) = L_a$.

Soundness of the analysis then amounts to:

Theorem 6.2 Assume that $\emptyset \vdash b : A \ \& \ P$ and

$$[pi_0 \mapsto b] \Longrightarrow_{ps_1}^{a_1} \ldots \Longrightarrow_{ps_k}^{a_k} PB$$

and let $penv$ be a mapping from process identifiers to elements of **LabSet** respecting the above derivation sequence and such that $penv(pi_0) = L_0$. We then have

$$\mathcal{R}^*(\text{COUNT}^{penv}[(a_1, ps_1), \cdots, (a_k, ps_k)]) \sqsubseteq (P \oplus [L_0 \mapsto A])$$

where $\mathcal{R}^*(C)(L_a)(L) = (\mathcal{R}(c), \mathcal{R}(i), \mathcal{R}(o), \mathcal{R}(f))$ whenever $C(L_a)(L) = (c, i, o, f)$. □

Note that the lefthand side of the inequality counts the number of operations for all processes whose labels is given (by L_a); hence our information is useful for static processor allocation.

To prove the theorem we need the following lemma expressing the sequential soundness of the analysis:

Lemma 6.3 If $\emptyset \vdash b : A \ \& \ P$ and $b \Rightarrow^p b'$ then there exists A_0, P_0, A' and P' such that $\emptyset \vdash p : A_0 \ \& \ P_0$, $\emptyset \vdash b' : A' \ \& \ P'$ and $A_0 \oplus A' \sqsubseteq A$ as well as $P_0 \oplus P' \sqsubseteq P$. □

Here we have extended the predicate of Table 5 to configurations by taking

$$\emptyset \vdash \surd : [\ ] \ \& \ [\ ]$$

To prove the concurrent soundness of the analysis we define

$$\vdash_{penv} PB : P$$

to mean that $PB = [pi_1 \mapsto b_1, \cdots, pi_j \mapsto b_j]$, $\emptyset \vdash b_1 : A_1 \ \& \ P_1, \cdots, \emptyset \vdash b_j : A_j \ \& \ P_j$ and $P = P_1 \oplus \cdots \oplus P_j \oplus [L_1 \mapsto A_1] \oplus \cdots \oplus [L_j \mapsto A_j]$ where $penv(pi_1) = L_1$, $\cdots$, $penv(pi_j) = L_j$. We then have the following lemma from which Theorem 6.2 immediately follows:

Proposition 6.4 Assume that $\vdash_{penv} PB : P$ and

$$PB \Longrightarrow_{ps_1}^{a_1} \ldots \Longrightarrow_{ps_k}^{a_k} PB'$$

where *penv* is a mapping from process identifiers to elements of **LabSet** respecting the above derivation sequence. Then there exists P' such that $\vdash_{penv} PB' : P'$ and

$$\mathcal{R}^*(\mathrm{COUNT}^{penv}[(a_1, ps_1), \cdots, (a_k, ps_k)]) \oplus P' \sqsubseteq P. \qquad \square$$

To obtain an efficient implementation of the analysis it is once more profitable to generate an equation system. This is hardly any different from the approach of Section 5 except that by now there is even greater scope for decomposing the flow variables into families of flow variables over simpler value spaces.

7 Dynamic Processor Allocation

The idea behind the dynamic processor allocation is that the decision of how to place processes on processors is taken dynamically. Again we will be interested in knowing which requirements this put on the processor but in contrast to the previous section we are only concerned with a single process rather than all processes with a given label. We shall now modify the analysis of Section 6 to associate *worst-case* information with the *process labels* rather than accumulating the total information. For each process label L_a we therefore ask the four questions of Section 4 taking the *maximum* information over all processes with label L_a: how many times are channels labelled by L created, how many times do channels labelled by L participate in input, how many times do channels labelled by L participate in output, and how many times are processes labelled by L generated?

Example 7.1 Let us return to the `pipe` function of Example 2.1 and assume that we want *dynamic processor allocation*. This means that all the processes labeled π *need not* reside on the same processor. The analysis should therefore estimate the *maximal* requirements of the instances of these processes as follows:

main program:	L_1:	`M` channels created
	π:	`M` processes created
process π:	L_1:	`M` inputs performed
	L_2:	`M` outputs performed
	L:	`I` output performed

Note that now we do record that each individual process labelled by π actually only communicates over L at most once.

Returning to the processor network of Example 6.1 we may allocate the main program on **P1** and the remaining processes on **P2** and **P3** (and possibly **P1** as well): say `f1` and `f3` on **P2** and `f2` and `id` on **P3**. Facilities for multitasking are needed on **P2** and **P3** and facilities for multiplexing on all of **P1**, **P2** and **P3**. $\square$

The inference system still has formulae

$$benv \vdash b : A \;\&\; P$$

where A and P are as in Section 6 and now *benv* is as in Section 4: it does not incorporate the P component[5].

[5]It could be as in Section 6 as well because we now combine P components using $\sqcup$ rather than $\oplus$.

$$benv \vdash \epsilon : [\,] \;\&\; [\,]$$

$$benv \vdash L!t : [L \mapsto (\mathtt{o},\mathtt{o},\mathtt{i},\mathtt{o})] \;\&\; [\,] \qquad benv \vdash L?t : [L \mapsto (\mathtt{o},\mathtt{i},\mathtt{o},\mathtt{o})] \;\&\; [\,]$$

$$benv \vdash t\ \textsc{chan}_L : [L \mapsto (\mathtt{i},\mathtt{o},\mathtt{o},\mathtt{o})] \;\&\; [\,] \qquad \frac{benv \vdash b : A \;\&\; P}{benv \vdash \textsc{fork}_L\ b : [L \mapsto (\mathtt{o},\mathtt{o},\mathtt{o},\mathtt{i})] \;\&\; ([L \mapsto A] \sqcup P)}$$

$$\frac{benv \vdash b_1 : A_1 \;\&\; P_1 \quad benv \vdash b_2 : A_2 \;\&\; P_2}{benv \vdash b_1; b_2 : A_1 \oplus A_2 \;\&\; P_1 \sqcup P_2} \qquad \frac{benv \vdash b_1 : A_1 \;\&\; P_1 \quad benv \vdash b_2 : A_2 \;\&\; P_2}{benv \vdash b_1 + b_2 : A_1 \sqcup A_2 \;\&\; P_1 \sqcup P_2}$$

$$\frac{benv[\beta \mapsto A] \vdash b : A \;\&\; P}{benv \vdash \textsc{rec}\ \beta.\, b : A \;\&\; P} \qquad benv \vdash \beta : A \;\&\; [\,] \quad \text{if } benv(\beta) = A$$

Table 6: Analysis for dynamic process allocation

A difference from Section 6 is that now we need to keep track of the individual process identifiers. We therefore redefine the function COUNT^{penv} as follows:

$$\mathrm{COUNT}^{penv}(\mathcal{X}) = \lambda L_a.\lambda L.((CC_{PI}(\mathcal{X},L), CI_{PI}(\mathcal{X},L), CO_{PI}(\mathcal{X},L), CF_{PI}(\mathcal{X},L)) \quad \text{where } PI = penv^{-1}(L_a))$$

where

- $CC_{PI}(\mathcal{X},L)$: the maximum over all $pi \in PI$ of the number of elements of the form $(t\ \textsc{chan}_L, pi)$ in $\mathcal{X}$,
- $CI_{PI}(\mathcal{X},L)$: the maximum over all $pi \in PI$ of the number of elements of the form $(L'!t?L, (pi', pi))$ in $\mathcal{X}$,
- $CO_{PI}(\mathcal{X},L)$: the maximum over all $pi \in PI$ of the number of elements of the form $(L!t?L', (pi, pi'))$ in $\mathcal{X}$, and
- $CF_{PI}(\mathcal{X},L)$: the maximum over all $pi \in PI$ of the number of elements of the form $(\textsc{fork}_L b, (pi, pi'))$ in $\mathcal{X}$.

Soundness of the analysis then amounts to:

Theorem 7.2 Assume that $\emptyset \vdash b : A \;\&\; P$ and

$$[pi_0 \mapsto b] \Longrightarrow^{a_1}_{ps_1} \ldots \Longrightarrow^{a_k}_{ps_k} PB$$

and let $penv$ be a mapping from process identifiers to elements of **LabSet** respecting the above derivation sequence and such that $penv(pi_0) = L_0$. We then have

$$\mathcal{R}^*(\mathrm{COUNT}^{penv}[(a_1, ps_1), \cdots, (a_k, ps_k)]) \sqsubseteq (P \sqcup [L_0 \mapsto A]).$$

where $\mathcal{R}^*$ is as in Theorem 6.2. □

Note that the lefthand side of the inequality gives the maximum number of operations over all processes with a given label; hence our information is useful for dynamic processor allocation.

To prove this result we need the following lemma expressing the sequential soundness of the analysis:

Lemma 7.3 If $\emptyset \vdash b : A \;\&\; P$ and $b \Rightarrow^{p} b'$ then there exists A_0, P_0, A' and P' such that $\emptyset \vdash p : A_0 \;\&\; P_0$, $\emptyset \vdash b' : A' \;\&\; P'$ and $A_0 \oplus A' \sqsubseteq A$ as well as $P_0 \sqcup P' \sqsubseteq P$. □

Here we have once more extended the predicate of Table 6 to configurations by taking

$$\emptyset \vdash \surd : [\,] \;\&\; [\,]$$

To prove the concurrent soundness of the analysis we need to associate information with process identifiers rather than process labels. So we define

$$\vdash_{penv} PB : \mathcal{P}$$

to mean that $PB = [pi_1 \mapsto b_1, \cdots, pi_j \mapsto b_j]$, $\emptyset \vdash b_1 : A_1 \;\&\; P_1, \cdots, \emptyset \vdash b_j : A_j \;\&\; P_j$ and

$$\mathcal{P} = (((P_1 \sqcup \cdots \sqcup P_j) \circ penv) \setminus \{pi_1, \cdots, pi_j\}) \sqcup [pi_1 \mapsto A_1, \cdots, pi_j \mapsto A_j]$$

where $\mathcal{P} \setminus PI$ gives o on PI and otherwise acts as $\mathcal{P}$. We shall also need a version CT of the function COUNT that associates information with process identifiers:

$$\mathrm{CT}(\mathcal{X}) = \lambda pi.\lambda L.(CC_{\{pi\}}(\mathcal{X}, L), CI_{\{pi\}}(\mathcal{X}, L), CO_{\{pi\}}(\mathcal{X}, L), CF_{\{pi\}}(\mathcal{X}, L))$$

We then have the following proposition from which we will be able to prove Theorem 7.2:

Proposition 7.4 Assume that $\vdash_{penv} PB : \mathcal{P}$ and

$$PB \Longrightarrow^{a_1}_{ps_1} \ldots \Longrightarrow^{a_k}_{ps_k} PB'$$

where $penv$ is a mapping from process identifiers to elements of **LabSet** respecting the above derivation sequence. Then there exists $\mathcal{P}'$ such that $\vdash_{penv} PB' : \mathcal{P}'$ and

$$\mathcal{R}^*(\mathrm{CT}[(a_1, ps_1), \cdots, (a_k, ps_k)]) \oplus \mathcal{P}' \sqsubseteq \mathcal{P}.$$ □

To obtain an efficient implementation of the analysis it is once more profitable to generate an equation system and the remarks at the end of the previous section still apply.

8 Conclusion

The specifications of the analyses for static and dynamic allocation have much in common; the major difference of course being that for static processor allocation we *accumulate* the total numbers whereas for dynamic processor allocation we calculate the *maximum*; a minor difference being that for the static analysis it was crucial to let behaviour environments include the P component whereas for the dynamic analysis this was hardly of any importance. Naturally, the proofs of their soundness differ in an important aspect: for the static analysis we could simply count the number of events for each label set whereas for the dynamic analysis the proof had to consider each of the process identifiers separately and then take the least upper bound (or maximum) of the number of events over all process identifiers with the same label set. This difference in proof technology is reminiscent of the difference between the formulation of MFP-style and MOP-style analyses: in the former the effects of paths (corresponding to process identifiers with the same label set) are merged along the way whereas in the latter the paths (corresponding to the process identifiers) have to be kept separate and their effect can only be merged when the propagation of effects have taken place.

Acknowledgements We would like to thank Torben Amtoft for many interesting discussions. This research has been funded in part by the LOMAPS (ESPRIT BRA) and DART (Danish Science Research Council) projects.

References

[1] J.Cai, R.Paige: Program Derivation by Fixed Point Computation. *Science of Computer Programming* **11**, pp. 197–261, 1989.

[2] R. Cridlig, E.Goubault: Semantics and analysis of Linda-based languages. *Proc. Static Analysis*, Springer Lecture Notes in Computer Science **724**, 1993.

[3] C.E.McDowell: A practical algorithm for static analysis of parallel programs. *Journal of parallel and distributed computing* **6**, 1989.

[4] A.Giacalone, P.Mishra, S.Prasad: Operational and Algebraic Semantics for Facile: a Symmetric Integration of Concurrent and Functional Programming. *Proc. ICALP'90*, Springer Lecture Notes in Computer Science **443**, 1990.

[5] K.Havelund, K.G.Larsen: The Fork Calculus. *Proc. ICALP'93*, Springer Lecture Notes in Computer Science **700**, 1993.

[6] M.S.Hecht: *Flow Analysis of Computer Programs*, North-Holland, 1977.

[7] Y.-C.Hung, G.-H.Chen: Reverse reachability analysis: a new technique for deadlock detection on communicating finite state machines. *Software — Practice and Experience* **23**, 1993.

[8] S.Jagannathan, S.Week: Analysing stores and references in a parallel symbolic language. *Proc. L&FP*, 1994.

[9] M.Jourdan, D.Parigot: Techniques for Improving Grammar Flow Analysis. *Proc. ESOP'90*, Springer Lecture Notes in Computer Science **432**, pp. 240–255, 1990.

[10] N. Mercouroff: An algorithm for analysing communicating processes. *Proc. of MFPS*, Springer Lecture Notes in Computer Science **598**, 1992.

[11] F.Nielson, H.R.Nielson: From CML to Process Algebras. *Proc. CONCUR'93*, Springer Lecture Notes in Computer Science **715**, 1993.

[12] H.R.Nielson, F.Nielson: Higher-Order Concurrent Programs with Finite Communication Topology. *Proc. POPL'94*, pp. 84–97, ACM Press, 1994.

[13] F.Nielson, H.R.Nielson: Constraints for Polymorphic Behaviours for Concurrent ML. *Proc. CCL'94*, Springer Lecture Notes in Computer Science **845**, 1994.

[14] J.H.Reif, S.A.Smolka: Dataflow analysis of distributed communicating processes. *International Journal of Parallel Programs* **19**, 1990.

[15] J.R.Reppy: Concurrent ML: Design, Application and Semantics. Springer Lecture Notes in Computer Science **693**, pp. 165–198, 1993.

[16] R.Tarjan: Iterative Algorithms for Global Flow Analysis. In J.Traub (ed.), *Algorithms and Complexity*, pp. 91–102, Academic Press, 1976.

[17] B.Thomsen. Personal communication, May 1994.

A Proofs

Proof of Lemma 4.3 The functionality of A is $E \to \mathbf{Abs}$ for a finite subset E of **LabSet** that includes all labels of b; hence $E \to \mathbf{Abs}$ is a complete lattice just as **Abs** is. We then proceed by structural induction upon b.

The base cases ϵ, $L!t$, $L?t$, and t $\mathtt{CHAN}_L$ are immediate as the sets in question are all singletons that do not depend on $benv$. The base case β is also immediate because the set is a singleton that depends on $benv$ in a straightforward way.

The case $\mathtt{FORK}_L\ b$ follows from the induction hypothesis because the set $\{A \mid benv \vdash \mathtt{FORK}_L\ b : A\}$ is obtained from the set $\{A \mid benv \vdash b : A\}$ by pointwise application of the function $\lambda A.A \oplus [L \mapsto (\mathtt{O}, \mathtt{O}, \mathtt{O}, \mathtt{I})]$ that is monotone and non-reductive (i.e. the result is greater than or equal to the argument). The cases $b_1; b_2$ and $b_1 + b_2$ follow from the induction hypothesis because $\oplus$ and $\sqcup$ are monotone and non-reductive.

For the case $\mathtt{REC}\beta.b$ let $\mathcal{B}_l^{benv}$ be the function that maps A_β to the least A_l such that $benv[\beta \mapsto A_\beta] \vdash b : A_l$ and similarly let $\mathcal{B}_g^{benv}$ be the function that maps A_β to the greatest A_g such that $benv[\beta \mapsto A_\beta] \vdash b : A_g$. By the induction hypothesis these functions exist and are monotone. Thus the set $\{A \mid benv \vdash \mathtt{REC}\beta.b : A\}$ will contain the least fixed point $\mathrm{LFP}(\mathcal{B}_l^{benv})$ of $\mathcal{B}_l^{benv}$ as its least element, and it will contain the greatest fixed point $\mathrm{GFP}(\mathcal{B}_g^{benv})$ of $\mathcal{B}_g^{benv}$ as its greatest element, and in particular the set will not be empty. That the set is monotone in $benv$ boils down to considering $benv_1 \sqsubseteq benv_2$ and showing $\mathrm{GFP}(\mathcal{B}_g^{benv_1}) \sqsubseteq \mathrm{GFP}(\mathcal{B}_g^{benv_2})$ and $\mathrm{LFP}(\mathcal{B}_l^{benv_1}) \sqsubseteq \mathrm{LFP}(\mathcal{B}_l^{benv_2})$ and this is a consequence of

$\mathcal{B}_g^{benv_1} \sqsubseteq \mathcal{B}_g^{benv_2}$ and $\mathcal{B}_l^{benv_1} \sqsubseteq \mathcal{B}_l^{benv_2}$ as follows from the induction hypothesis. □

Proof of Lemma 4.5 We proceed by induction on the inference of $b \Rightarrow^p b'$.

The base cases $p \Rightarrow^p \epsilon$, $\epsilon \Rightarrow^\epsilon \surd$ and $b \Rightarrow^\epsilon b$ are immediate since $A \oplus [\,] = [\,] \oplus A = A$.

The case $\textsc{rec}\beta.b \Rightarrow^\epsilon b[\beta \mapsto \textsc{rec}\beta.b]$ follows from

Fact A.1 If $benv[\beta \mapsto A_0] \vdash b : A$ and $benv \vdash b_0 : A_0$ then $benv \vdash b[\beta \mapsto b_0] : A$.

The case $b_1; b_2 \Rightarrow^p b_1'; b_2$ because $b_1 \Rightarrow^p b_1'$ follows from the induction hypothesis and the associativity and monotonicity of $\oplus$. The case $b_1; b_2 \Rightarrow^p b_2$ because $b_1 \Rightarrow^p \surd$ follows from the induction hypothesis, the monotonicity of $\oplus$ and that O is the identity for $\oplus$.

The case $b_1 + b_2 \Rightarrow^p b_i'$ because $b_i \Rightarrow^p b_i'$ (for $i = 1, 2$) follows from the induction hypothesis and that $\sqcup$ is a least upper bound operation. □

Proof of Proposition 4.6 We proceed by induction on the length k of the derivation sequence. The case $k = 0$ is trivial and for the induction step we inspect how the first step is performed.

First assume that $PB[pi \mapsto b] \Longrightarrow_{pi}^{a} PB[pi \mapsto b']$ because $b \Rightarrow^a b'$ (so we apply one of the first three rules of Table 2). Then $\vdash PB : A_1$ and $\emptyset \vdash b : A_2$ for some A_1 and A_2 with $A_1 \oplus A_2 = A$. Hence Lemma 4.5 gives $\emptyset \vdash a : A_0$ and $\emptyset \vdash b' : A_2''$ such that $A_0 \oplus A_2'' \sqsubseteq A_2$. The induction hypothesis gives

$$\mathcal{R}^*(\mathrm{COUNT}[a_2, \cdots, a_k]) \oplus A' \sqsubseteq A_1 \oplus A_2''$$

and since a will be one of ϵ and $t\ \textsc{chan}_{L_0}$ (for some t and L_0) we have $\mathcal{R}^*(\mathrm{COUNT}[a]) \sqsubseteq A_0$. From this we get

$$\begin{array}{lll} \mathcal{R}^*(\mathrm{COUNT}[a, a_2, \cdots, a_k]) \oplus A' & \sqsubseteq & \mathcal{R}^*(\mathrm{COUNT}[a]) \oplus \mathcal{R}^*(\mathrm{COUNT}[a_2, \cdots, a_k]) \oplus A' \\ & \sqsubseteq & A_0 \oplus A_1 \oplus A_2'' \\ & \sqsubseteq & A_1 \oplus A_2 = A \end{array}$$

Next assume that $PB[pi_1 \mapsto b] \Longrightarrow_{pi_1, pi_2}^{\mathrm{FORK}_{L_0} b_0} PB[pi_1 \mapsto b'][pi_2 \mapsto b_0]$ because $b \Rightarrow^{\mathrm{FORK}_{L_0} b_0} b'$. Then $\vdash PB : A_1$ and $\emptyset \vdash b : A_2$ for some A_1 and A_2 with $A_1 \oplus A_2 = A$. Hence Lemma 4.5 gives $\emptyset \vdash \textsc{fork}_{L_0} b_0 : A_0$ and $\emptyset \vdash b' : A_2''$ such that $A_0 \oplus A_2'' \sqsubseteq A_2$. Now $A_0 = [L_0 \mapsto (\mathsf{O}, \mathsf{O}, \mathsf{O}, \mathsf{I})] \oplus A_0''$ where $\emptyset \vdash b_0 : A_0''$. The induction hypothesis gives

$$\mathcal{R}^*(\mathrm{COUNT}[a_2, \cdots, a_k]) \oplus A' \sqsubseteq A_1 \oplus A_2'' \oplus A_0''$$

and since $\mathcal{R}^*(\mathrm{COUNT}[\textsc{fork}_{L_0} b_0]) \sqsubseteq [L_0 \mapsto (\mathsf{O}, \mathsf{O}, \mathsf{O}, \mathsf{I})]$ it follows that

$$\begin{array}{lll} \mathcal{R}^*(\mathrm{COUNT}[\textsc{fork}_{L_0} b_0, a_2, \cdots, a_k]) \oplus A' & \sqsubseteq & [L_0 \mapsto (\mathsf{O}, \mathsf{O}, \mathsf{O}, \mathsf{I})] \oplus A_1 \oplus A_2'' \oplus A_0'' \\ & \sqsubseteq & A_1 \oplus A_2'' \oplus A_0 \sqsubseteq A_1 \oplus A_2 = A \end{array}$$

Finally assume that $PB[pi_1 \mapsto b_1][pi_2 \mapsto b_2] \Longrightarrow_{pi_1, pi_2}^{L_1!t?L_2} PB[pi_1 \mapsto b_1'][pi_2 \mapsto b_2']$ because $b_1 \Rightarrow^{L_1!t} b_1'$ and $b_2 \Rightarrow^{L_2?t} b_2'$. Then $\vdash PB : A_1$, $\emptyset \vdash b_1 : A_{21}$ and $\emptyset \vdash b_2 : A_{22}$ for some A_1, A_{21} and A_{22} with $A_1 \oplus A_{21} \oplus A_{22} = A$. Hence Lemma 4.5 gives $\emptyset \vdash L_1!t : A_{01}$ and $\emptyset \vdash b_1' : A_{21}''$ such that $A_{01} \oplus A_{21}'' \sqsubseteq A_{21}$ and furthermore $\emptyset \vdash L_2?t : A_{02}$ and $\emptyset \vdash b_2' : A_{22}''$ such that $A_{02} \oplus A_{22}'' \sqsubseteq A_{22}$. Now $A_{01} = [L_1 \mapsto (\mathsf{O}, \mathsf{O}, \mathsf{I}, \mathsf{O})]$ and $A_{02} = [L_2 \mapsto (\mathsf{O}, \mathsf{I}, \mathsf{O}, \mathsf{O})]$. The induction hypothesis gives

$$\mathcal{R}^*(\mathrm{COUNT}[a_2,\cdots,a_k]) \oplus A' \sqsubseteq A_1 \oplus A''_{21} \oplus A''_{22}$$

and it follows that

$$\mathcal{R}^*(\mathrm{COUNT}[L_1!t?L_2,a_2,\cdots,a_k]) \oplus A' \sqsubseteq A_{01} \oplus A_{02} \oplus A_1 \oplus A''_{21} \oplus A''_{22} \sqsubseteq A$$

This completes the proof. □

Proof of Theorem 5.1 We proceed by structural induction on b. Without further mentioning we shall use the fact that the flow variables produced by $\mathcal{E}[\![B : \pi : b]\!]$ are either of the form $\langle \pi\pi' \rangle$ or of the form $\langle \beta' \rangle$ for some $\beta' \in EV(b)$.

The base cases ϵ, $L!t$, $L?t$ and t $\mathtt{CHAN}_L$ are immediate as the set of exposed variables is empty and the inference system and the equation system both have the same unique solution. The base case β is immediate because both sets equal $\{([\beta \mapsto A], A) \mid A \in E \to \mathbf{Abs}\}$.

For the base case $b_1; b_2$ we perform the following calculation:

$$\begin{aligned}
&\{\, (benv\lceil EV(b_1;b_2), A) \mid benv \vdash b_1;b_2 : A \,\} \\
&= \{\, ((benv_1\lceil EV(b_1)) \cup (benv_2\lceil EV(b_2)), A_1 \oplus A_2) \mid \\
&\qquad benv_1 \vdash b_1 : A_1,\ benv_2 \vdash b_2 : A_2, \\
&\qquad \forall \beta \in (EV(b_1) \cap EV(b_2)) : benv_1(\beta) = benv_2(\beta) \,\} \\
&= \{\, ((\sigma_1\lceil EV(b_1)) \cup (\sigma_2\lceil EV(b_2)), \sigma_1(\langle \pi 1 \rangle) \oplus \sigma_2(\langle \pi 2 \rangle)) \mid \\
&\qquad \sigma_1 \models \mathcal{E}[\![B : \pi 1 : b_1]\!],\ \sigma_2 \models \mathcal{E}[\![B : \pi 2 : b_2]\!], \\
&\qquad \forall \beta \in (EV(b_1) \cap EV(b_2)) : \sigma_1(\langle \beta \rangle) = \sigma_2(\langle \beta \rangle) \,\} \\
&= \{\, (\sigma\lceil EV(b_1;b_2), \sigma(\langle \pi \rangle)) \mid \sigma \models \mathcal{E}[\![B : \pi : b_1;b_2]\!] \,\}
\end{aligned}$$

where we have used the induction hypothesis and that the only flow variables common to $\mathcal{E}[\![B : \pi 1 : b_1]\!]$ and $\mathcal{E}[\![B : \pi 2 : b_2]\!]$ are $\{\, \langle \beta \rangle \mid \beta \in EV(b_1) \cap EV(b_2) \,\}$. The case $b_1 + b_2$ is similar and the case $\mathtt{FORK}_L\ b$ is along the same lines.

For the case $\mathtt{REC}\beta.b$ we first consider the effect of $\mathrm{CLOSE}^{\pi}_{\beta}$. For all sets $\mathbf{E}$ of flow equations containing the equation $(\langle \beta \rangle = \langle \pi \rangle)$ we have

$$\sigma \models \mathrm{CLOSE}^{\pi}_{\beta}(\mathbf{E}) \quad \text{iff} \quad \sigma[\langle \beta \rangle \mapsto \sigma(\langle \pi \rangle)] \models \mathbf{E}$$

We can then perform the calculations

$$\begin{aligned}
&\{\, (benv\lceil EV(\mathtt{REC}\beta.b), A) \mid benv \vdash \mathtt{REC}\beta.b : A \,\} \\
&= \{\, (benv_1\lceil EV(\mathtt{REC}\beta.b), A) \mid benv_1 \vdash b : A,\ benv_1(\beta) = A \,\} \\
&= \{\, (\sigma_1\lceil EV(\mathtt{REC}\beta.b), \sigma_1(\langle \pi 1 \rangle)) \mid \\
&\qquad \sigma_1 \models \mathcal{E}[\![B : \pi 1 : b]\!],\ \sigma_1(\langle \beta \rangle) = \sigma_1(\langle \pi 1 \rangle) \,\} \\
&= \{\, (\sigma\lceil EV(\mathtt{REC}\beta.b), \sigma(\langle \pi \rangle)) \mid \\
&\qquad \sigma \models \mathcal{E}[\![B : \pi 1 : b]\!] \cup \{\, \langle \beta \rangle = \langle \pi \rangle,\ \langle \pi \rangle = \langle \pi 1 \rangle \,\} \,\} \\
&= \{\, (\sigma\lceil EV(\mathtt{REC}\beta.b), \sigma(\langle \pi \rangle)) \mid \sigma \models \mathcal{E}[\![B : \pi : \mathtt{REC}\beta.b]\!] \,\}
\end{aligned}$$

thus finishing the proof. □

Proof of Lemma 6.3 This is a straightforward modification of the proof of Lemma 4.5. For the recursive case it uses

Fact A.2 If $benv[\beta \mapsto A_0 \ \& \ P_0] \vdash b : A \ \& \ P$ and $benv \vdash b_0 : A_0 \ \& \ P_0$ then $benv \vdash b[\beta \mapsto b_0] : A \ \& \ P$.

as a substitute for the Fact A.1. □

Proof of Proposition 6.4 As in the proof of Proposition 4.6 we proceed by induction on the length k of the derivation sequence. The case $k = 0$ is trivial and for the induction step we inspect how the first step is performed.

First assume that $PB[pi \mapsto b] \Longrightarrow^{a}_{pi} PB[pi \mapsto b']$ because $b \Rightarrow^{a} b'$ and assume $penv\ pi_1 = L$. Then $\vdash PB : P_1$ and $\emptyset \vdash b : A_2 \ \& \ P_2$ for some P_1, P_2 and A_2 with $P_1 \oplus P_2 \oplus [L \mapsto A_2] = P$. From Lemma 6.3 we get $\emptyset \vdash a : A_0 \ \& \ P_0$ and $\emptyset \vdash b' : A_2'' \ \& \ P_2''$ such that $A_0 \oplus A_2'' \sqsubseteq A_2$ as well as $P_0 \oplus P_2'' \sqsubseteq P_2$. The induction hypothesis gives

$$\mathcal{R}^*(\mathrm{COUNT}^{penv}[(a_2, ps_2), \cdots, (a_k, ps_k)]) \oplus P' \sqsubseteq P_1 \oplus P_2'' \oplus [L \mapsto A_2'']$$

and since a will be one of ϵ and t $\mathtt{CHAN}_{L_0}$ (for some t and L_0) we have

$$\mathcal{R}^*(\mathrm{COUNT}^{penv}[(a, pi)]) \sqsubseteq P_0 \oplus [L \mapsto A_0]$$

From this we get

$$\begin{array}{l}
\mathcal{R}^*(\mathrm{COUNT}^{penv}[(a, pi), (a_2, ps_2), \cdots, (a_k, ps_k)]) \oplus P' \\
\sqsubseteq \mathcal{R}^*(\mathrm{COUNT}^{penv}[(a, pi)]) \oplus \mathcal{R}^*(\mathrm{COUNT}^{penv}[(a_2, ps_2), \cdots, (a_k, ps_k)]) \oplus P' \\
\sqsubseteq P_0 \oplus [L \mapsto A_0] \oplus P_1 \oplus P_2'' \oplus [L \mapsto A_2''] \\
\sqsubseteq P_1 \oplus P_2 \oplus [L \mapsto A_2] \\
= P
\end{array}$$

Next assume that $PB[pi_1 \mapsto b] \Longrightarrow^{\mathrm{FORK}_{L_0} b_0}_{pi_1, pi_2} PB[pi_1 \mapsto b'][pi_2 \mapsto b_0]$ because $b \Rightarrow^{\mathrm{FORK}_{L_0} b_0} b'$ and assume $penv(pi_1) = L$ and $penv(pi_2) = L_0$. Then $\vdash PB : P_1$ and $\emptyset \vdash b : A_2 \ \& \ P_2$ for some P_1, P_2 and A_2 with $P_1 \oplus P_2 \oplus [L \mapsto A_2] = P$. From Lemma 6.3 we get $\emptyset \vdash \mathtt{FORK}_{L_0} b_0 : A_0 \ \& \ P_0$ and $\emptyset \vdash b' : A_2'' \ \& \ P_2''$ such that $A_0 \oplus A_2'' \sqsubseteq A_2$ as well as $P_0 \oplus P_2'' \sqsubseteq P_2$. Now $A_0 = [L_0 \mapsto (\mathtt{O}, \mathtt{O}, \mathtt{O}, \mathtt{I})]$ and $P_0 = [L_0 \mapsto A_0''] \oplus P_0''$ where $\emptyset \vdash b_0 : A_0'' \ \& \ P_0''$. The induction hypothesis gives

$$\begin{array}{l}
\mathcal{R}^*(\mathrm{COUNT}^{penv}[(a_2, ps_2), \cdots, (a_k, ps_k)]) \oplus P' \\
\sqsubseteq P_1 \oplus [L \mapsto A_2''] \oplus P_2'' \oplus [L_0 \mapsto A_0''] \oplus P_0''
\end{array}$$

and since

$$\mathcal{R}^*(\mathrm{COUNT}^{penv}[(\mathtt{FORK}_{L_0} b_0, (pi_1, pi_2))]) \sqsubseteq [L \mapsto A_0]$$

it follows that

$$
\begin{array}{l}
\mathcal{R}^*(\mathrm{COUNT}^{penv}[(\mathtt{FORK}_{L_0} b_0, (pi_1, pi_2)), (a_2, ps_2), \cdots, (a_k, ps_k)]) \oplus P' \\
\sqsubseteq [L \mapsto A_0] \oplus P_1 \oplus [L \mapsto A_2''] \oplus P_2'' \oplus [L_0 \mapsto A_0''] \oplus P_0'' \\
\sqsubseteq P_1 \oplus P_2'' \oplus [L \mapsto A_2] \oplus P_0 \\
\sqsubseteq P_1 \oplus P_2 \oplus [L \mapsto A_2] \\
= P
\end{array}
$$

Finally assume that $PB[pi_1 \mapsto b_1][pi_2 \mapsto b_2] \Longrightarrow^{L_1!t?L_2}_{pi_1,pi_2} PB[pi_1 \mapsto b_1'][pi_2 \mapsto b_2']$ because $b_1 \Rightarrow^{L_1!t} b_1'$ and $b_2 \Rightarrow^{L_2?t} b_2'$ and assume that $penv(pi_1) = L_1$ and $penv(pi_2) = L_2$. Then $\vdash PB : P_1$, $\emptyset \vdash b_1 : A_{21} \;\&\; P_{21}$ and $\emptyset \vdash b_2 : A_{22} \;\&\; P_{22}$ for some P_1, P_{21}, P_{22}, A_{21} and A_{22} with $P_1 \oplus P_{21} \oplus P_{22} \oplus [L_1 \mapsto A_{21}] \oplus [L_2 \mapsto A_{22}] = P$. Hence Lemma 6.3 gives $\emptyset \vdash L_1!t : A_{01} \;\&\; P_{01}$ and $\emptyset \vdash b_1' : A_{21}'' \;\&\; P_{21}''$ such that $A_{01} \oplus A_{21}'' \sqsubseteq A_{21}$ and $P_{01} \oplus P_{21}'' \sqsubseteq P_{21}$. Furthermore $\emptyset \vdash L_2?t : A_{02} \;\&\; P_{02}$ and $\emptyset \vdash b_2' : A_{22}'' \;\&\; P_{22}''$ such that $A_{02} \oplus A_{22}'' \sqsubseteq A_{22}$ and $P_{02} \oplus P_{22}'' \sqsubseteq P_{22}$. Now $A_{01} = [L_1 \mapsto (\mathsf{o}, \mathsf{o}, \mathsf{i}, \mathsf{o})]$ and $A_{02} = [L_2 \mapsto (\mathsf{o}, \mathsf{i}, \mathsf{o}, \mathsf{o})]$. We have

$$
\mathcal{R}^*(\mathrm{COUNT}^{penv}[(L_1!t?L_2, (pi_1, pi_2))]) \sqsubseteq [L_1 \mapsto A_{01}] \oplus P_{01} \oplus [L_2 \mapsto A_{02}] \oplus P_{02}
$$

The induction hypothesis gives

$$
\begin{array}{l}
\mathcal{R}^*(\mathrm{COUNT}^{penv}[(a_2, ps_2), \cdots, (a_k, ps_k)]) \oplus P' \\
\sqsubseteq P_1 \oplus P_{21}'' \oplus [L_1 \mapsto A_{21}''] \oplus P_{22}'' \oplus [L_2 \mapsto A_{22}'']
\end{array}
$$

and it follows that

$$
\begin{array}{l}
\mathcal{R}^*(\mathrm{COUNT}^{penv}[(L_1!t?L_2, (pi_1, pi_2)), (a_2, ps_2), \cdots, (a_k, ps_k)]) \oplus P' \\
\sqsubseteq \mathcal{R}^*(\mathrm{COUNT}^{penv}[(L_1!t?L_2, (pi_1, pi_2))]) \oplus \\
\qquad\qquad \mathcal{R}^*(\mathrm{COUNT}^{penv}[(a_2, ps_2), \cdots, (a_k, ps_k)]) \oplus P' \\
\sqsubseteq [L_1 \mapsto A_{01}] \oplus P_{01} \oplus [L_2 \mapsto A_{02}] \oplus P_{02} \oplus P_1 \oplus P_{21}'' \oplus [L_1 \mapsto A_{21}''] \oplus P_{22}'' \oplus [L_2 \mapsto A_{22}''] \\
\sqsubseteq [L_1 \mapsto A_{21}] \oplus P_{21} \oplus [L_2 \mapsto A_{22}] \oplus P_{22} \oplus P_1 \\
= P
\end{array}
$$

This completes the proof. □

Proof of Lemma 7.3 This is a straightforward modification of the proofs of Lemmas 4.5 and 6.3 using that $\sqcup$ is a least upper bound operation on a complete lattice with o as least element. For the recursive case it uses

Fact A.3 If $benv[\beta \mapsto A_0] \vdash b : A \;\&\; P$ and $benv \vdash b_0 : A_0 \;\&\; P_0$ then there exists P' such that $benv \vdash b[\beta \mapsto b_0] : A \;\&\; P'$ and $P' \sqsubseteq P_0 \sqcup P$.

as a substitute for Facts A.1 and A.2. □

Proof of Proposition 7.4 As in the proof of Propositions 4.6 and 6.4 we proceed by induction on the length k of the derivation sequence. The case $k = 0$ is trivial and for the induction step we inspect how the first step is performed.

First assume that $PB[pi \mapsto b] \Longrightarrow^{a}_{pi} PB[pi \mapsto b']$ because $b \Rightarrow^{a} b'$. Then $\vdash_{penv} PB : \mathcal{P}_1$ and $\emptyset \vdash b : A_2 \;\&\; P_2$ for some $\mathcal{P}_1$, P_2 and A_2 with

$$
(\mathcal{P}_1 \setminus \{pi\}) \sqcup ((P_2 \circ penv) \setminus (PI \cup \{pi\})) \sqcup [pi \mapsto A_2] = \mathcal{P}
$$

where $PI = Dom(PB)$. From Lemma 7.3 we get $\emptyset \vdash a : A_0 \;\&\; P_0$ and $\emptyset \vdash b' : A_2'' \;\&\; P_2''$ such that $A_0 \oplus A_2'' \sqsubseteq A_2$ as well as $P_0 \sqcup P_2'' \sqsubseteq P_2$. The induction hypothesis gives

$$\begin{array}{l}\mathcal{R}^*(\mathrm{CT}[(a_2, ps_2), \cdots, (a_k, ps_k)]) \oplus \mathcal{P}' \\ \sqsubseteq (\mathcal{P}_1 \setminus \{pi\}) \sqcup ((P_2'' \circ penv) \setminus (PI \cup \{pi\})) \sqcup [pi \mapsto A_2'']\end{array}$$

and since a will be one of ϵ and t $\mathtt{CHAN}_{L_0}$ (for some t and L_0) we have

$$\mathcal{R}^*(\mathrm{CT}[(a, pi)]) \sqsubseteq [pi \mapsto A_0]$$

From this we get

$$\begin{array}{l}\mathcal{R}^*(\mathrm{CT}[(a, pi), (a_2, ps_2), \cdots, (a_k, ps_k)]) \oplus \mathcal{P}' \\ \sqsubseteq \mathcal{R}^*(\mathrm{CT}[(a, pi)]) \oplus \mathcal{R}^*(\mathrm{CT}[(a_2, ps_2), \cdots, (a_k, ps_k)]) \oplus \mathcal{P}' \\ \sqsubseteq [pi \mapsto A_0] \oplus ((\mathcal{P}_1 \setminus \{pi\}) \sqcup ((P_2'' \circ penv) \setminus (PI \cup \{pi\})) \sqcup [pi \mapsto A_2'']) \\ \sqsubseteq (\mathcal{P}_1 \setminus \{pi\}) \sqcup ((P_2'' \circ penv) \setminus (PI \cup \{pi\})) \sqcup [pi \mapsto A_0 \oplus A_2''] \\ \sqsubseteq (\mathcal{P}_1 \setminus \{pi\}) \sqcup ((P_2 \circ penv) \setminus (PI \cup \{pi\})) \sqcup [pi \mapsto A_2] \\ = \mathcal{P}\end{array}$$

Next assume that $PB[pi_1 \mapsto b] \Longrightarrow^{\mathrm{FORK}_{L_0} b_0}_{pi_1, pi_2} PB[pi_1 \mapsto b'][pi_2 \mapsto b_0]$ because $b \Rightarrow^{\mathrm{FORK}_{L_0} b_0} b'$ and note that $penv(pi_2) = L_0$. Then $\vdash_{penv} PB : \mathcal{P}_1$ and $\emptyset \vdash b : A_2 \;\&\; P_2$ for some $\mathcal{P}_1$, P_2 and A_2 with

$$(\mathcal{P}_1 \setminus \{pi_1\}) \sqcup ((P_2 \circ penv) \setminus (PI \cup \{pi_1\})) \sqcup [pi_1 \mapsto A_2] = \mathcal{P}$$

where $PI = Dom(PB)$. From Lemma 7.3 we get $\emptyset \vdash \mathtt{FORK}_{L_0} b_0 : A_0 \;\&\; P_0$ and $\emptyset \vdash b' : A_2'' \;\&\; P_2''$ such that $A_0 \oplus A_2'' \sqsubseteq A_2$ as well as $P_0 \sqcup P_2'' \sqsubseteq P_2$. Now $A_0 = [L_0 \mapsto (\mathtt{O}, \mathtt{O}, \mathtt{O}, \mathtt{I})]$ and $P_0 = [L_0 \mapsto A_0''] \sqcup P_0''$ where $\emptyset \vdash b_0 : A_0'' \;\&\; P_0''$. The induction hypothesis gives

$$\begin{array}{l}\mathcal{R}^*(\mathrm{CT}[(a_2, ps_2), \cdots, (a_k, ps_k)]) \oplus \mathcal{P}' \\ \sqsubseteq (\mathcal{P}_1 \setminus \{pi_1, pi_2\}) \sqcup (((P_2'' \sqcup P_0'') \circ penv) \setminus (PI \cup \{pi_1, pi_2\})) \sqcup [pi_1 \mapsto A_2'', pi_2 \mapsto A_0'']\end{array}$$

and since

$$\mathcal{R}^*(\mathrm{CT}[(\mathtt{FORK}_{L_0} b_0, (pi_1, pi_2))]) \sqsubseteq [pi_1 \mapsto A_0]$$

it follows that

$$\begin{array}{l}\mathcal{R}^*(\mathrm{CT}[(\mathtt{FORK}_{L_0} b_0, (pi_1, pi_2)), (a_2, ps_2), \cdots, (a_k, ps_k)]) \oplus \mathcal{P}' \\ \sqsubseteq [pi_1 \mapsto A_0] \oplus ((\mathcal{P}_1 \setminus \{pi_1, pi_2\}) \sqcup (((P_2'' \sqcup P_0'') \circ penv) \setminus (PI \cup \{pi_1, pi_2\})) \\ \qquad\qquad \sqcup [pi_1 \mapsto A_2'', pi_2 \mapsto A_0'']) \\ \sqsubseteq (\mathcal{P}_1 \setminus \{pi_1, pi_2\}) \sqcup (((P_2'' \sqcup P_0'') \circ penv) \setminus (PI \cup \{pi_1, pi_2\})) \sqcup [pi_1 \mapsto A_0 \oplus A_2'', pi_2 \mapsto A_0''] \\ \sqsubseteq (\mathcal{P}_1 \setminus \{pi_1, pi_2\}) \sqcup ((P_2 \circ penv) \setminus (PI \cup \{pi_1, pi_2\}))) \sqcup [pi_1 \mapsto A_2, pi_2 \mapsto A_0''] \\ \sqsubseteq (\mathcal{P}_1 \setminus \{pi_1\}) \sqcup ((P_2 \circ penv) \setminus (PI \cup \{pi_1\})) \sqcup [pi_1 \mapsto A_2] \\ = \mathcal{P}\end{array}$$

where we have used that $A_0'' \sqsubseteq (P_0 \circ penv)(pi_2) \sqsubseteq (P_2 \circ penv)(pi_2) \sqsubseteq \mathcal{P}_1(pi_2) \sqcup (P_2 \circ penv)(pi_2)$.

Finally assume that $PB[pi_1 \mapsto b_1][pi_2 \mapsto b_2] \Longrightarrow^{L_1!t?L_2}_{pi_1,pi_2} PB[pi_1 \mapsto b_1'][pi_2 \mapsto b_2']$ because $b_1 \Rightarrow^{L_1!t} b_1'$ and $b_2 \Rightarrow^{L_2?t} b_2'$. Then $\vdash_{penv} PB : \mathcal{P}_1$, $\emptyset \vdash b_1 : A_{21} \,\&\, P_{21}$ and $\emptyset \vdash b_2 : A_{22} \,\&\, P_{22}$ for some $\mathcal{P}_1$, P_{21}, P_{22}, A_{21} and A_{22} with

$$(\mathcal{P}_1 \setminus \{pi_1, pi_2\}) \sqcup (((P_{21} \sqcup P_{22}) \circ penv) \setminus (PI \cup \{pi_1, pi_2\})) \sqcup [pi_1 \mapsto A_{21}, pi_2 \mapsto A_{22}] = \mathcal{P}$$

where $PI = Dom(PB)$. Hence Lemma 7.3 gives $\emptyset \vdash L_1!t : A_{01} \,\&\, P_{01}$ and $\emptyset \vdash b_1' : A_{21}'' \,\&\, P_{21}''$ such that $A_{01} \oplus A_{21}'' \sqsubseteq A_{21}$ and $P_{01} \sqcup P_{21}'' \sqsubseteq P_{21}$. Furthermore $\emptyset \vdash L_2?t : A_{02} \,\&\, P_{02}$ and $\emptyset \vdash b_2' : A_{22}'' \,\&\, P_{22}''$ such that $A_{02} \oplus A_{22}'' \sqsubseteq A_{22}$ and $P_{02} \sqcup P_{22}'' \sqsubseteq P_{22}$. Now $A_{01} = [L_1 \mapsto (\mathtt{O}, \mathtt{O}, \mathtt{I}, \mathtt{O})]$ and $A_{02} = [L_2 \mapsto (\mathtt{O}, \mathtt{I}, \mathtt{O}, \mathtt{O})]$. We have

$$\mathcal{R}^*(\mathrm{CT}[(L_1!t?L_2, (pi_1, pi_2))]) \sqsubseteq [pi_1 \mapsto A_{01}, pi_2 \mapsto A_{02}]$$

The induction hypothesis gives

$$\begin{array}{l}\mathcal{R}^*(\mathrm{CT}[(a_2, ps_2), \cdots, (a_k, ps_k)]) \oplus \mathcal{P}' \\ \sqsubseteq (\mathcal{P}_1 \setminus \{pi_1, pi_2\}) \sqcup (((P_{21}'' \sqcup P_{22}'') \circ penv) \setminus (PI \cup \{pi_1, pi_2\})) \sqcup [pi_1 \mapsto A_{21}'', pi_2 \mapsto A_{22}'']\end{array}$$

and it follows that

$$\begin{array}{l}\mathcal{R}^*(\mathrm{CT}[(L_1!t?L_2, (pi_1, pi_2)), (a_2, ps_2), \cdots, (a_k, ps_k)]) \oplus \mathcal{P}' \\ \sqsubseteq \mathcal{R}^*(\mathrm{CT}[(L_1!t?L_2, (pi_1, pi_2))]) \oplus \mathcal{R}^*(\mathrm{CT}[(a_2, ps_2), \cdots, (a_k, ps_k)]) \oplus \mathcal{P}' \\ \sqsubseteq [pi_1 \mapsto A_{01}, pi_2 \mapsto A_{02}] \oplus ((\mathcal{P}_1 \setminus \{pi_1, pi_2\}) \sqcup (((P_{21}'' \sqcup P_{22}'') \circ penv) \setminus (PI \cup \{pi_1, pi_2\})) \\ \qquad\qquad \sqcup [pi_1 \mapsto A_{21}'', pi_2 \mapsto A_{22}'']) \\ \sqsubseteq (\mathcal{P}_1 \setminus \{pi_1, pi_2\}) \sqcup (((P_{21} \sqcup P_{22}) \circ penv) \setminus (PI \cup \{pi_1, pi_2\})) \sqcup [pi_1 \mapsto A_{21}, pi_2 \mapsto A_{22}] \\ = \mathcal{P}\end{array}$$

This completes the proof. □

Proof of Theorem 7.2 Assume $\emptyset \vdash b : A \,\&\, P$ and

$$[pi_0 \mapsto b] \Longrightarrow^{a_1}_{ps_1} \ldots \Longrightarrow^{a_k}_{ps_k} PB$$

From $\emptyset \vdash b : A \,\&\, P$ we get

$$\vdash_{penv} [pi_0 \mapsto b] : ((P \circ penv) \setminus \{pi_0\}) \sqcup [pi_0 \mapsto A]$$

From Proposition 7.4 we therefore get

$$\mathcal{R}^*(\mathrm{CT}[(a_1, ps_1), \cdots, (a_k, ps_k)]) \sqsubseteq ((P \circ penv) \setminus \{pi_0\}) \sqcup [pi_0 \mapsto A]$$

We clearly have $((P \circ penv) \setminus \{pi_0\}) \sqcup [pi_0 \mapsto A] \sqsubseteq (P \sqcup [L_0 \mapsto A]) \circ penv$ since $penv(pi_0) = L_0$ so for all L

$$\bigsqcup \{\mathcal{R}^*(\mathrm{CT}[(a_1, ps_1), \cdots, (a_k, ps_k)])(pi) \mid penv(pi) = L\} \sqsubseteq (P \sqcup [L_0 \mapsto A])(L)$$

From the definition of COUNT^{penv} and CT it follows that

$$\begin{aligned}&\mathrm{COUNT}^{penv}[(a_1, ps_1), \cdots, (a_k, ps_k)](L)\\ &= \max\{CT[(a_1, ps_1), \cdots, (a_k, ps_k)](pi) \mid penv(pi) = L\}\end{aligned}$$

where max is applied componentwise. Using the "additivity" of $\mathcal{R}^*$ we now get

$$\begin{aligned}&\bigsqcup\{\mathcal{R}^*(\mathrm{CT}[(a_1, ps_1), \cdots, (a_k, ps_k)])(pi) \mid penv(pi) = L\}\\ &= \mathcal{R}^*(\max\{\mathrm{CT}[(a_1, ps_1), \cdots, (a_k, ps_k)])(pi) \mid penv(pi) = L\})\\ &= \mathcal{R}^*(\mathrm{COUNT}^{penv}[(a_1, ps_1), \cdots, (a_k, ps_k)](L))\end{aligned}$$

This completes the proof. □