McGill University
School of Computer Science
Sable Research Group

Points-to Inspired Static Analysis for the π-Calculus

Sable Technical Report No. 2003-02

Sam B. Sanjabi, Clark Verbrugge

www.sable.mcgill.ca

Abstract

We present a flow and context insensitive control flow analysis of the π-calculus which computes a superset of the names that can possibly be transmitted over each channel. The analysis is a simpler version of the constraint-based analysis of [6] based on techniques inspired from points-to analyis, but requires no extension to the basic calculus. The algorithm computes the same solution as the approach in [6], but with better space efficiency. Our approach is closely related to techniques used in compiler optimizers to do points-to analyses, and is a simpler presentation, provides insight into the inner workings of the process , and immediately suggests other possible analyses on π-calculus processes.

1 Introduction

Static analysis is used to ascertain properties that a program will hold in all possible executions. These properties will hold independently of the input to the program and the context in which it is run.

In this paper we provide a simple and space-optimal formulation of static analysis as applied to the π-calculus. Such analyses have been used to verify properties of concurrent programs, and in particular security properties of abstract protocols. Our algorithm is derived from an understanding of "points-to" or alias analysis techniques used in compiler optimization to compute identity relations in the context of indirect access, and so demonstrates a strong connection between these problem domains.

The π-calculus [9] is a concurrent process algebra introduced by Milner and Parrow. It provides a framework for the study of parallelism in a formal setting, and has been studied extensively. Static analysis on the π-calculus should provide results that could be used in establishing formal security properties for concurrent systems.

Contributions of this paper include:

- A demonstrated relation between "traditional" compiler-driven points-to analysis and an analysis of the channel passing behaviour of π-calculus processes. This suggests other points-to analysis techniques may also be applicable or informative.
- A simpler, easy to understand (or implement) algorithm for computing flow information on π-calculus processes than is currently available. Unlike existing techniques, our analysis requires no extensions, syntactic or otherwise, to the π-calculus.
- Since our algorithm runs in space $\Theta(n^2)$ we also reduce the space requirements of the existing constraint-based solution system, achieving optimal space efficiency without any sacrifice in time complexity.

2 Overview

Section 3 gives a survey of research related to the work at hand. Section 4 provides an overview of the π-calculus and the constraint based analysis of [5]. Section 5 and 6 develop our algorithm from an intuition inspired by points-to analysis. Section 7 presents pseudo-code for our algorithm based on the flow equations derived from the intuitive model. Section 8 runs through a complete example on a π-calculus process. Section 9 provides a proof of correctness, and presents a complexity analysis. Section 10, makes concluding remarks and discusses possible variations of our analysis for future work.

3 Related Work

Static program analysis [11] computes information about the execution of a program at compile-time. Traditionally, it has been used in optimizations of the machine code generated by a language compiler. In such cases, the properties ascertained may have been the detection of redundant memory operations [7] or compile-time type determination in languages with references [14]. One such property is the approximation of the possible run-time resolution of pointers through points-to analysis [8, 16, 1, 17, 3].

Such analysis techniques have also been applied on process algebrae in more formal settings. Nielson *et al* use an analysis on mobile ambients to show how abstract interpretation can be expressed in a constraint-based formalism [12]. Other static analyses have been developped on Idealized Algol [15] and the π-calculus [6].

Program analysis techniques have recently been used to establish provable security properties of sequential and concurrent computer programs. Most of these are either constraint-based or derived type systems. Bodei *et al* check secrecy and non-interference properties by statically analyzing the spi-calculus [4]. Nielson *et al* use a static analysis to validate firewalls formalized as mobile ambients [13]. Malacaria and Hankin show an analysis based on abstract interpretation that calculates a notion of information flow in a sequential program [15] based on game semantics. A static analysis on the π-calculus [9] presented by Bodei *et al* detects information leaks in concurrent systems [5].

Several static analysis techniques have been developed for analyzing concurrent programs. These tend to focus on problems specific to compiler optimization; e.g., Naumovich et al's May-Happen-in-Parallel (MHP) analysis for Java [10], or Aldrich et al's work on synchronization removal [2].

4 Required Background

4.1 The π-calculus

I will be using the monadic, synchronous, match-free calculus with standard *early* operational semantics as the base language on which my analysis is performed. This is simply because it is the same language used in [5] without the extension of channels and binders, and also without the use of the 'match' prefix (which I have removed for simplicity, but this does not hinder expressiveness).

Here I'll recall the syntax of the π-calculus model that we'll be using.

Definition 4.1 *Let $\mathcal{N}$ be an infinite set of names ranged over by a, b, ..., x, y, ... and let τ be a distinguished symbol such that $\mathcal{N} \cap \{\tau\} = \emptyset$. A π-calculus process is built according to the syntax*

$$P ::= \mathbf{0} \mid x(y).P \mid \bar{x}y.P \mid \tau.P \mid P + P \mid P|P \mid (\nu x)P \mid !P$$

The standard notions of structural congruence ($\equiv$) and reduction ($\rightarrow$) are used to make up the complete language definition. Note that this definition does not include any extensions to Milner's original definition. I'll use the symbol $\stackrel{\alpha}{\equiv}$ to denote α congruence.

4.2 A Constraint Based Analysis

Here we briefly recall the control flow analysis given in [5], which is based on a constraint resolution model. The algorithm is based on the above language extended with *channels* and *binders* which are used to track the passage of names. The syntax of the extended calculus is as in definition 4.1 with the exception that the bound name of an input prefix is extended with a *binder* and the bound name in a restriction is extended with a *channel*. The syntax of the annotated calculus is defined as follows:

Definition 4.2 *Let $\mathcal{N}$ be an infinite set of names ranged over by a, b, ..., x, y, ... and let τ be a distinguished symbol such that $\mathcal{N} \cap \{\tau\} = \emptyset$. Let $\mathcal{B}$ be a non-empty set of binders ranged over by ranged over by β,β',...; and let $\mathcal{C}$ be a non-empty set of channels ranged over by χ,χ',... such that $\mathcal{B} \cap \mathcal{C} = \emptyset$. An (annotated) π-calculus process is built according to the syntax*

$$P ::= \mathbf{0} \mid x(y^{\beta}).P \mid \bar{x}y.P \mid \tau.P \mid P + P \mid P|P \mid (\nu x^{\chi})P \mid !P$$

The operational semantics used in this language are effectively identical to those used by the standard calculus, and are given in [5]. The notable difference is that the *Match* rule is omitted. As we'll see below, the channels and binders are only used to distinguish between repeated names for distinct channels.

Given an (annotated) process P, a set of constraints is generated according to the function given in table 1. The association between names and *channels/binders* is maintained by the *marker environment* (*me*), which is only updated when *channels* and *binders* are declared. This function is identical to table 3 given in [5] except that there is no entry for the 'match' prefix. Consequently, there is no need to carry the set C (which is only changed for the match prefix). Thus the constraints are greatly simplified.

The least solution to these constraints (for a given process P) yields a pair of functions (ρ_P, κ_P), abbreviated as $(\rho, \kappa)_P$, the first of which maps a *binder* (or input names) to a set of *channels* (or restricted names) whose values they could take, and the second takes a *channel* to a set of *channels* that could be transmitted over the corresponding name.

5 Eliminating Extensions

A brief inspection of table 1 reveals that the marker environment (*me*) is only updated for the input prefix and restriction. Thus, if there are no repeated names for distinct channels (a state that can be achieved α-conversion), then *me* is simply a bijective function which associates each name to a *channel* or a *binder* upon its declaration. This result is summarized by the following proposition:

Proposition 5.1 *Given an annotated π-calculus process P, there exists a process P' such that $P \stackrel{\alpha}{\equiv} P'$, and the marker environment (me) of $\mathcal{G}[\![P']\!]_{me}$ is bijective.*

$$
\begin{array}{lcl}
\mathcal{G}[\![\mathbf{0}]\!]_{me} & = & \emptyset \\
\mathcal{G}[\![\tau.P]\!]_{me} & = & \mathcal{G}[\![P]\!]_{me} \\
\mathcal{G}[\![\bar{x}y.P]\!]_{me} & = & \mathcal{G}[\![P]\!]_{me} \cup \\
 & & \{(\{\{\chi\} \subseteq \rho(me(x))\}) \Rightarrow \rho(me(y)) \subseteq \kappa(\chi) \mid \chi \in \mathcal{C}\} \\
\mathcal{G}[\![x(y^{\beta}).P]\!]_{me} & = & \mathcal{G}[\![P]\!]_{me[y \mapsto \beta]} \cup \\
 & & \{(\{\{\chi\} \subseteq \rho(me(x))\}) \Rightarrow \kappa(\chi) \subseteq \rho(\beta) \mid \chi \in \mathcal{C}\} \\
\mathcal{G}[\![P_1 + P_2]\!]_{me} & = & \mathcal{G}[\![P_1]\!]_{me} \cup \mathcal{G}[\![P_2]\!]_{me} \\
\mathcal{G}[\![P_1 \mid P_2]\!]_{me} & = & \mathcal{G}[\![P_1]\!]_{me} \cup \mathcal{G}[\![P_2]\!]_{me} \\
\mathcal{G}[\![(\nu x^{\chi})P]\!]_{me} & = & \mathcal{G}[\![P]\!]_{me[x \mapsto \chi]} \\
\mathcal{G}[\![!P]\!]_{me} & = & \mathcal{G}[\![P]\!]_{me}
\end{array}
$$

Table 1: Constraint generation for the annotated π-calculus

Proof Given P, construct P' by α-converting P such that there are no repeated names. Now consider the set of constraints $\mathcal{G}[\![P']\!]_{me}$. Since me is only updated during name binding (via an input prefix or restriction), and since there are no repeated names, each binding will map a distinct name to a distinct *channel* or *binder*.$\square$

Thus the α-conversion eliminates the need for an extension to the calculus completely as it removes the ambiguity in names (which can now be used in place of channels and binders).

6 Some Intuition

In this section I provide an intuitive description of the algorithm based on Andersen's technique for doing a flow insensitive points-to analysis in the `C` programming language [3]. This is an inclusion based analysis which, for every reference variable (pointer) x in a `C` program, computes a superset of the variables that it could point to during the execution of the program. Consider the following example program fragment:

```
1:  int *p, *q, *r, n;
2:  while (cond) {
3:      p = q;
4:      q = r;
5:      r = &n;
6:  }
```

The first step in Andersen's analysis is to strip out only the relevant statements in the program (i.e., only those statements that affect assignments to pointers). Thus, only the statements in lines 3, 4, and 5 are considered. Next, the analysis iterates through these statements: for each pointer assignment, the program learns a new constraint that it must satisfy. Let `pts(x)` be the set of variables that a pointer variable `x` could point to and initialize this set to $\emptyset$ for each pointer in the program. Then, in the first iteration of this example, the program first considers the statement assigning `q` to `p`, from which it learns that `pts`(`q`) $\subseteq$ `pts`(`p`). Thus it must add everything in `pts(q)` (which is empty at this point) to `pts(p)`. The same happens when the algorithm considers statement 4. When it arrives at the next statement (`r = &n;`), the deduction is made that the pointer `r` points to the variable `n`, thus `n` should be added to `pts(r)`. The situation can be expressed as a graph with variables as nodes and a directed arc $a \rightarrow b$ representing inclusion of b in `pts(a)` in the points to set. The iterations of Andersen's analysis on our program fragment are represented as such a graph in figure 1.

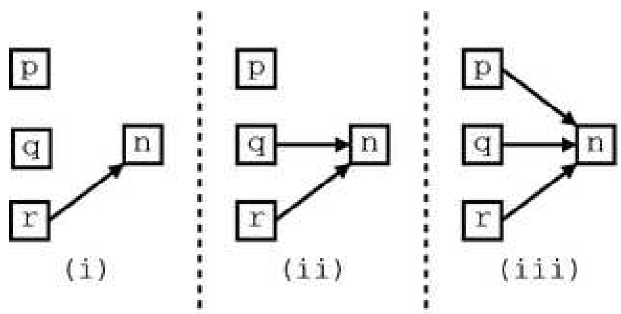

Figure 1: The three iterations of Andersen't analysis on our program fragment

The algorithm continues to iterate through the relevant statements of the program until a fixed point is reached (i.e., until none of the points-to sets change in a given iteration). Once the fixed point is reached, every pointer element will contain `n` in its points-to set (and thus, `p`, `q`, and `r` will all have an arc to `n` in the graph). Notice that for any `C` program, the graph created by this procedure will be a DAG (Directed Acyclic Graph) because single pointers can only point to variables, double pointers (pointers to pointers) can only point to single pointers, and so on. This is the only respect in which our control flow analysis for the π-calculus will differ from Andersen's points-to analysis for `C`.

In order to see how we can adapt this technique to the π-calculus, I'll first informally define an unusual partition of the names of a π-calculus process. Let $names(P)$ be the set of names of a π-calculus process P, then I define $\mathbf{bi}(P)$ as the set of names of P that are *b*ound by an *i*nput prefix (formal definition given in table 2). Then, for any given name $x \in names(P) \setminus \mathbf{bi}(P)$, we (informally) define $\mathbf{out}(x)$ as the set of names that could be *transmitted* over x during the reduction of P (via an output prefix). Next for any name $x \in \mathbf{bi}(P)$ we define $\mathbf{in}(x)$ as the set of names that could be *received* by x during the reduction of P (via and input prefix). Now, as in the points-to example above, we can provide an iterative procedure that will update these sets.

The first step in our procedure is to α-convert our process to avoid any name conflicts as outlined above. Next, analogous to considering only pointer assignments from `C` programs, we consider *only the set of input and output prefixes in our process.* We iterate through these prefixes adding names to the **in** and **out** sets of each name as required. For instance, consider the information gained when reaching an input prefix $x(y)$, where $x \notin \mathbf{bi}(P)$. In such a case, we know that x is not an

$\mathbf{bi}(\mathbf{0})$	$\triangleq$	$\emptyset$	$\mathbf{bi}(P+Q)$	$\triangleq$	$\mathbf{bi}(P) \cup \mathbf{bi}(Q)$
$\mathbf{bi}(x(y).P)$	$\triangleq$	$\{y\} \cup \mathbf{bi}(P)$	$\mathbf{bi}(P \vert Q)$	$\triangleq$	$\mathbf{bi}(P) \cup \mathbf{bi}(Q)$
$\mathbf{bi}(\bar{x}y.P)$	$\triangleq$	$\mathbf{bi}(P)$	$\mathbf{bi}((\nu x)P)$	$\triangleq$	$\mathbf{bi}(P)$
$\mathbf{bi}(\tau.P)$	$\triangleq$	$\mathbf{bi}(P)$	$\mathbf{bi}(!P)$	$\triangleq$	$\mathbf{bi}(P)$

Table 2: Formal definition of $\mathbf{bi}(P)$

input-bounded name, and hence couldn't represent any other name than itself, hence we would simply like to add any names that could possibly be transmitted over x ($\mathbf{out}(x)$) to the set of names that could be received by y ($\mathbf{in}(y)$). This reasoning gives us our first flow equation:

$$\mathbf{in}(y) \leftarrow \mathbf{out}(x) \cup \mathbf{in}(y) \ for\ x(y)\ such\ that\ x \notin \mathbf{bi}(P)$$

We can similarly arrive at such an equation for five other possible cases, comprising the case of an input prefix where the channel name *is* bound by input, and four cases involving an output prefix and the possible ways that its two names could be bound. An enumeration of all six cases follows:

1. input prefix with channel name not bound by input
2. input prefix with channel name bound by input
3. output prefix with both the emitting channel and emitted name bound by input
4. output prefix with the emitting channel bound by input but the emitted name not bound by input
5. output prefix with the emitting channel not bound by input but the emitted name bound by input
6. output prefix with neither the emitting channel nor the emitted name bound by input

These cases, as well as the flow equations they represent are directly coded into the formal definition of our algorithm given in table 3 below. This situtation can be represented as a graph (see example in section 8) just as the points-to analysis case could, where in this case a node in the graph is a channel name, and an arc $x \rightarrow y$ from a node $x \in \mathbf{bi}(P)$ denotes that $y \in \mathbf{in}(x)$, and a node $x \rightarrow y$ from a node $x \notin \mathbf{bi}(P)$ denotes that $y \in \mathbf{out}(x)$.

7 The Full Algorithm

Let $\mathbf{bi}(P)$ be defined as above, and for a given P, we define a pair of functions $\mathbf{in}_P : \mathcal{N} \rightarrow \mathscr{P}(\mathcal{N})$ and $\mathbf{out}_P : \mathcal{N} \rightarrow \mathscr{P}(\mathcal{N})$. The first function is intended to track the possible names that a name bound by input could receive throughout the course of the computation of P, and the second is intended to track the set of names that can be transmitted (via an output prefix) over the given channel name. As we shall see below, these functions nearly correspond exactly to the functions ρ and κ (from [5]) respectively. To make this correspondence exact, we adopt the convention that

INPUT: process P
1: α-convert P to avoid name repetition
2: Initialize **in** and **out** functions to $\perp$
3: Strip out all of the input and output prefixes and iterate through them
4: foreach input prefix $x(y)$ do
5: if $x \in \mathbf{bi}(P)$ then
6: $\mathbf{in}(y) \leftarrow \mathbf{out}(\mathbf{in}(x)) \cup \mathbf{in}(y)$
7: else
8: $\mathbf{in}(y) \leftarrow \mathbf{out}(x) \cup \mathbf{in}(y)$
9: foreach output prefix $\bar{x}y$ do
10: if $x \in \mathbf{bi}(P) \land y \in \mathbf{bi}(P)$ then
11: $\forall z \in \mathbf{in}(x), \mathbf{out}(z) \leftarrow \mathbf{in}(y) \cup \mathbf{out}(\mathbf{in}(x))$
12: if $x \in \mathbf{bi}(P) \land y \notin \mathbf{bi}(P)$ then
13: $\forall z \in \mathbf{in}(x), \mathbf{out}(z) \leftarrow \{y\} \cup \mathbf{out}(\mathbf{in}(x))$
14: if $x \notin \mathbf{bi}(P) \land y \in \mathbf{bi}(P)$ then
15: $\mathbf{out}(x) \leftarrow \mathbf{in}(y) \cup \mathbf{out}(x)$
16: if $x \notin \mathbf{bi}(P) \land y \notin \mathbf{bi}(P)$ then
17: $\mathbf{out}(x) \leftarrow \{y\} \cup \mathbf{out}(x)$
18: Iterate 4-17 until you reach a fixed point.

Table 3: An iterative control flow analysis of the π-calculus

$\mathbf{in}_P(x) = x$ if $x \notin \mathbf{bi}(P)$ and that $\mathbf{out}_P(x) = \emptyset$ if $x \in \mathbf{bi}(P)$. We generalize the **out** functions to sets in the obvious way, i.e. for a set $S \subseteq \mathcal{N}$ define $\mathbf{out}(S)$ as

$$\mathbf{out}(S) = \bigcup_{x \in S} \mathbf{out}(x)$$

Given a π-calculus process P, the algorithm computes a pair of functions ($\mathbf{in}_P$,$\mathbf{out}_P$) - abbreviated $(\mathbf{in}, \mathbf{out})_P$ which provide, for each name in P a set of names for which it could be substituted during the execution of P. The algorithm presented in table 3 computes these functions following the intuition presented in the previous section: for each input/output prefix, the function simply adds all possible names that could be substituted for the input or output name to that name's **in** or **out** set (based on what is known about these sets in the current iteration).

8 A Complete Example

Here we'll illustrate our algorithm with an example. Consider the following π-calculus process:

$$P \equiv a(x).(\nu b)(\nu c)((\bar{b}a.\bar{x}x.b(y).\bar{y}c + \bar{b}d.\bar{a}c) \mid b(z).\bar{b}z) \mid d(w)$$

This is identical to example 3.1 in [5] for comparison, except that the extensions have been removed and the process has been α-converted as above in order to eliminate ambiguities in naming.

Note that this process has limited reduction opportunities. However, because our analysis is flow insensitive, it will necessarily give a less precise conservative approximation.

We can calculate that $\mathbf{bi}(P) = \{x, y, z, w\}$ and that $names(P) \setminus \mathbf{bi}(P) = \{a, b, c, d\}$. Analogous to Andersen's analysis where all statements are stripped of the program except for pointer assignments, we consider only input and output prefixes and iterate through them. Our list of such prefixes for this process is $(a(x), b(y), b(z), d(w))$ for the input prefixes and $(\bar{b}a, \bar{x}x, \bar{y}c, \bar{b}d, \bar{a}c, \bar{b}z)$ for the output prefixes. We can now initialize the sets **in** and **out** to $\emptyset$ for each name of P, and iteratively make our way through the list updating the values. Note that we could go through the prefixes in any order and still obtain the same solution (although perhaps not in the same number of iterations).

In the first iteration, our procedure first hits the input prefix $a(x)$. Since a is not bound by input (i.e. $\notin \mathbf{bi}(P)$), we know that $\mathbf{in}(a)$ is irrelevant, thus we would need to update $\mathbf{in}(x)$ with the set of possible values that a could assume by having a name transmitted over it, which by definition is $\mathbf{out}(a)$ (which is currently empty). Thus, no new information is gained from the prefix $a(x)$, and similarly for the other input prefixes. The output prefixes begin with $\bar{b}a$, which by line 17 yields that $\mathbf{out}(b) = \{a\}$ since neither a nor b are bound by input. The next prefix $(\bar{x}x)$ does not yield any new data because $x \in \mathbf{bi}(P)$ and $\mathbf{in}(x) = \emptyset$, similarly for the prefixes $\bar{y}c$ and $\bar{b}z$ since we have no information about the input bound names y and z. The prefixes $\bar{b}d$ and $\bar{a}c$, however, do tell us that $\mathbf{out}(b) \supseteq \{d\}$ and $\mathbf{out}(a) \supseteq \{c\}$ respectively. Hence, at the end of the first iteration, we have that $\mathbf{out}(b) = \{a, d\}$ and $\mathbf{out}(a) = \{c\}$. This situation is presented as a graph in figure 2(i).

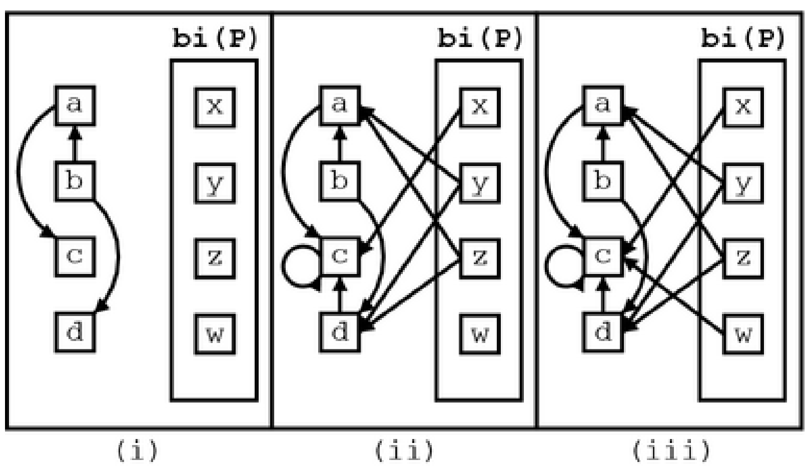

Figure 2: The three iterations of our π-calculus analysis of P

The second iteration begins again by going through the list of input prefixes, but this time, there is more information available than $\perp$. The prefix $a(x)$ at this point tells us that anything transmitted over a must be part of the names received by x, thus $\mathbf{in}(x) = \{c\}$ after seeing this prefix because $\mathbf{out}(a) = \{c\}$. Similarly, the prefixes $b(y)$ and $b(z)$ assign everything in the **out** set of b to the **in** set of z yielding $\mathbf{in}(y) = \{a, d\}$ and $\mathbf{in}(z) = \{a, d\}$. The last input prefix $d(w)$ yields no new information because we have no knowledge of any names transmitted by d. No output prefix with both names unbound by input will have any effect now because their direct effect was already accounted for in the first iteration, this leaves $\bar{x}x$, $\bar{y}c$, and $\bar{b}z$. The first indicates that anything that x may have received (which is currently $\mathbf{in}(x) = \{c\}$) could be transmitted over itself, thus we get $\mathbf{out}(c) = \{c\}$. The second says that c could be transmitted over anything that y may have received $(\mathbf{in}(y) = \{a, d\})$, hence we only add c to $\mathbf{out}(d)$ since $c \in \mathbf{out}(a)$ already. The third prefix

($\bar{b}z$) says that anything that z may have received could be transmitted over b, so all elements of $\mathbf{in}(z) = \{a, d\}$ must be added to $\mathbf{out}(b)$. But a and d are already in this set so we don't need to do anything. The situation after this iteration is presented graphically in figure 2(ii).

Proceeding in a similar way, we find that the third iteration of the process only adds the element c to $\mathbf{in}(w)$, and the fourth iteration makes no change indicating that we have reached a fixed point. The resulting graph representing the final solution of our analysis on P is shown in figure 2(iii).

Note that for a name y in $\mathbf{bi}(P)$, the set $\mathbf{in}(y)$ will consist only of names that are not in $\mathbf{bi}(P)$. The fact that the function **in** is defined over all names above is simply a convention used to compare our result with those computed in [5]. This is simply because y must have been defined in an input prefix $x(y)$, and we have defined our flow equations to be fully closed to transmitted names (i.e. we only add such names to our sets). However, names that aren't bound by input may have arcs to any other name not bound by input. Thus, while this graph does not end up becoming a DAG, we do have a bipartite graph with $\mathbf{bi}(P)$ a source subset (and hence no path exists to exit the subset $names(P) \setminus \mathbf{bi}(P)$. This strong intuitive relationship between points-to analysis in `C` and control flow analysis in the π-calculus allows us to fully understand the exact workings of our analysis, which we formally discuss in the next section.

9 Analysis

Here we present the proof of our algorithm's correctness by showing that its solution is equivalent to the solution from [5]. The longer proofs can be found in the appendix.

9.1 Correctness

First observe that the solution space of our algorithm - that is, pairs of functions (**in**,**out**) as defined above - can be partially ordered by pointwise inclusion, and forms a complete lattice as in [6]. The following lemma states that this solution space is isomorphic to the solution space using in the constraint based method:

Lemma 9.1 *Let* $(\mathbf{in}, \mathbf{out})$ *be the family of pairs of functions as defined above, and let* (ρ, κ) *be the family of paris of functions defined in [6], then* $(\mathbf{in}, \mathbf{out}) \simeq (\rho, \kappa)$.

Proof Since all of the functions **in**,**out**,ρ, and κ take countably infinite sets to the powersets of countably infinite sets, and by the fact that the pairs are ordered by pointwise inclusion, an isomorphism clearly exists. □

An immediate corollary of this result is that $(\mathbf{in}, \mathbf{out})$ is a Moore family (because (ρ, κ) is [5]), and hence the existence of solutions is given by Bodei *et al*'s result. We denote the partial order relation in this lattice with $\sqsubseteq$.

Now, let $\mathcal{S} = (\mathbf{in}, \mathbf{out})$ be our set of proposed solutions, we can view lines 4-17 of the algorithm given in table 3 above as a function $f_P : \mathcal{S} \to \mathcal{S}$ on this set because (given a process P), the algorithm's only effect is on the **in** and **out** sets of the names in the process. Notably, every possible branch that f_P can take involves adding elements to at least one of the sets (or doing nothing), thust f_P is certainly monotone. Thus, elementary fixed point theory tells us that f_P will reach a fixed point when iterated from bottom, which in this case is the element $(\perp, \perp)$ comprised of a pair of functions that take every input to the empty set.

For a given process P, let $(\mathbf{in}, \mathbf{out})_P$ be the least fixed point of f_P in $\mathcal{S}$. Let $\hat{P}$ be the same process as P annotated with *channels* and *binders*, and let $(\rho, \kappa)_{\hat{P}}$ be the least solution of the constraint set $\mathcal{G}[\![\hat{P}]\!]_{me}$. Our main result can now be expressed as a consequence of the following lemmas, the first of which establishes that the solution computed by the iterative algorithm is a valid solution to the constraint based approach (up to the isomorphism me). Letting $\circ$ stand for function composition, and abbreviating $(f \circ \mathbf{in} \circ g, f \circ \mathbf{out} \circ g)_P$ to $f \circ (\mathbf{in}, \mathbf{out})_P \circ g$, we use the bijective marker environment to compare our results to those of the constraint based model:

Lemma 9.2 *Given a process P, $me^{-1} \circ (\mathbf{in}, \mathbf{out})_P \circ me$ satisfies the constraints $\mathcal{G}[\![\hat{P}]\!]_{me}$.*

Proof See appendix A.

Corollary 9.3 *Given a process P, $me^{-1} \circ (\mathbf{in}, \mathbf{out})_P \circ me \sqsupseteq (\rho, \kappa)_{\hat{P}}$*

Proof Follows immediately from the facts that $(\rho, \kappa)_{\hat{P}}$ is the least solution to the constraint set. $\square$

The next lemma shows that the iterative solution is in fact at least as good as the constraint based solution, effectively establishing the equivalence of the two approaches:

Lemma 9.4 *Given a process P, $me^{-1} \circ (\mathbf{in}, \mathbf{out})_P \circ me \sqsubseteq (\rho, \kappa)_{\hat{P}}$.*

Proof See appendix A.

With these preliminary results, we can easily deduce our conclusion that the iterative and constraint based methods are equivalent up to the isomorphism me:

Theorem 9.5 $me^{-1} \circ (\mathbf{in}, \mathbf{out})_P \circ me = (\rho, \kappa)_{\hat{P}}$

Proof The result follows immediately from lemma 9.4 and corollary 9.3.$\square$

Having established that the two solutions are equivalent, the correctness of our algorithm follows from the results in [5].

9.2 Complexity

Expressing the running time in terms of the number N of symbols in the process, we first note that the number of *names* in the process is certainly less than N. Now observe that we have $O(N)$ in and out sets to compute (one for each name), and we add in the worst case a single name to only one of the sets in each iteration (if we add no names to any set then we are at a fixed point). Thus if we run until all names are in every set (in the worst case) then we run at most $O(N^2)$

iterations of the outer loop. Each of the inner **foreach** loops can be run at most $O(N)$ times as this is the maximum number of input and output prefices that can exist in a given process definition. Each branch in the inner loops is run once per iteration, and these are constant time operations except for the computation of $\mathbf{out}(\mathbf{in}(x))$ which is a merge of at most $O(N)$ sets of size O(N), an operation which is $\Theta(N^2)$. This yields a total time complexity of $O(N^5)$, the same as Bodei's constraint system.

However, our algorithm only ever handles a maximum of $O(N)$ sets each with a maximum of $O(N)$ elements, and so only requires the optimal $\Theta(N^2)$ space complexity, a reduction of the $O(N^3)$ space needed by the constraint system.

10 Conclusions and Future Work

We presented an iterative version of Bodei et al's control flow analysis for the π-calculus which clearly shows the workings of the analysis, and draws heavy inspiration from Andersen's points-to analysis for `C`. Furthermore, we have given an argument that allows us to conduct the analysis without augmenting the calculus in any way.

An immediate variation of this analysis would be to come up with a possibly faster algorithm by using Steensgaard's [17] technique for doing points-to analysis in "almost" linear time. This involves, in type constraint language, an *equality* based analysis in which constraints are formulated as equivalence relations rather than subsets. This technique led to a considerable speed-up in points-to analysis algorithms (nearly two orders of magnitude), but also a reduction in the accuracy of the discovered solutions. However, the apparent lack of disjoint sets in the π-calculus setting is a hurdle that will have to be overcome before a similar performance benefit can be achieved.

In the other extreme, we can imagine flow/context sensitive analyses that account for choice and parallel composition operators in the calculus. A flow sensitive constraint set generated as in table 1 may need to remove some constraints when encountering these operators. Such an analysis would almost certainly be more inefficient, and may not even provide a deterministic answer in the general case if the choice operator is also considered, but the accuracy of its solutions may prove benefitial in applications such as security.

References

[1] A.Aiken, M.Fhndrich, and J.S.Foster. Flow-insensitive points-to analysis with term and set constraints. Technical Report 97-964, University of California, Berkeley, Computer Science Division, Aug 1997.

[2] Jonathan Aldrich, Emin Sirer, Craig Chambers, and Susan J. Eggers. Comprehensive synchronization elimination for Java. *Science of Computer Programming*, 47(2–3):91–120, 2003.

[3] Lars Ole Andersen. *Program Analysis And Specialization for the C Programing Language.* PhD thesis, May 1999.

[4] C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Static analysis for secrecy and non-interference in networks of processes. *Lecture Notes in Computer Science*, 2127, 2001.

[5] C.Bodei, P.Degano, F.Nielson, and H. Riis Nielson. Static analysis for the pi-calculus with application to security. *INFCTRL: Information and Computation (formerly Information and Control*, 168, 2001.

[6] C.Bodei, P.Degano, F.Nielson, and H.Riis Nielson. Control flow analysis for the π-calculus. In *Proceedings of CONCUR '98*, volume 1466 of *Lecture Notes In Computer Science*, pages 84–98. Springer-Verlag, 1998.

[7] K. Cooper and L. Xu. An efficient static analysis algorithm to detect redundant memory operations, 2002.

[8] Jeffrey S. Foster, Manuel Fhndrich, and Alexander Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Static Analysis Symposium 2000*, Santa Barbara, California, June 2000.

[9] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, 1992.

[10] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proceedings of the joint 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 338–354, September 1999.

[11] F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[12] Flemming Nielson, Rene Rydhof Hansen, and Hanne Riis Nielson. Abstract interpretation of mobile ambients. *Science of Computer Programming*, 47:145–175, 2003.

[13] Flemming Nielson, Hanne Riis Nielson, Rene Rydhof Hansen, and Jacob Grydholt Jensen. Validating firewalls in mobile ambients. In *International Conference on Concurrency Theory*, pages 463–477, 1999.

[14] Hemant D. Pande and Barbara G. Ryder. Static type determination for C++. In *C++ Conference*, pages 85–98, 1994.

[15] P.Malacaria and C.Hankin. Non-deterministic games and program analysis: An application to security. In *Proceedings of the 14th Annual IEEE Symposium on Logic In Computer Science*, pages 443–452. IEEE Computer Society Press, 1999.

[16] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '2001)*, October 2001.

[17] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

A Proofs

Proof of Lemma 9.2 Assuming an appropriately α-converted process, me^{-1} is certainly well defined by proposition 5.1. Observe that $me^{-1} \circ (\mathbf{in}, \mathbf{out})_P \circ me$ is a pair of functions of the same type as $(\rho, \kappa)_{\hat{P}}$ and thus can be compared. For clarity, I will abbreviate common function compositions as follows:

$$\begin{aligned} \hat{\mathbf{in}} &\stackrel{def}{=} (me^{-1} \circ \mathbf{in} \circ me) \\ \hat{\mathbf{out}} &\stackrel{def}{=} (me^{-1} \circ \mathbf{out} \circ me) \end{aligned}$$

I also omit the P and $\hat{P}$ subscripts as they are implicit. Furthermore, since constraints are only added to $\mathcal{G}[\![P]\!]_{me}$ for input and output prefixes (and because our algorithm does not consider any other parts of the process), we only need to consider the role of these prefixes in determining our solution to f_P. We do this by analysis on the same set of cases used to generate the algorithm, we elaborate the first subcase in some detail, but remain brief on the rest as they all use similar tricks:

(Case 1: Input Prefix $x(y)$) In this case the constraint set generated by $\mathcal{G}$ is

$$\{(\{\{\chi\} \subseteq \rho(me(x))\}) \Rightarrow \kappa(\chi) \subseteq \rho(\beta) \mid \chi \in \mathcal{C}\}$$

Now we must divide this into subcases where x is bound by input or not (as y is bound by input in each case regardless):

[Subcase 1a: $x \in \mathbf{bi}(P)$ **]**

Suppose that $\{\chi\} \subseteq \hat{\mathbf{in}}(me(x))$, then by line 6 in our algorithm we have that

$$\mathbf{out}(\mathbf{in}(x)) \subseteq \mathbf{in}(y) \tag{1}$$

because of our fixed point iteration. We also have that for any x

$$\begin{aligned} \mathbf{in}(x) = &\quad \mathbf{in}(me^{-1}(me(x))) &&, \text{by bijectivity of } me \\ = &\quad (me^{-1} \circ \mathbf{in})(me(x)) &&, \text{definition of composition} \end{aligned} \tag{2}$$

Substituting 2 into 1 we obtain

$$\mathbf{out}((me^{-1} \circ \mathbf{in})(me(x))) \subseteq (me^{-1} \circ \mathbf{in})(me(y))$$

Using a method identical to that used in equation 2 on the **out** function, and the definition of $\hat{\mathbf{in}}$ we can obtain

$$(me^{-1} \circ \mathbf{out})(\hat{\mathbf{in}}(me(x))) \subseteq (me^{-1} \circ \mathbf{in})(me(y))$$

... and composing with me on both sides immediately yields

$$\hat{\mathbf{out}}(\hat{\mathbf{in}}(me(x)) \subseteq \hat{\mathbf{in}}(me(y)) \tag{3}$$

Now, since we've supposed that $\chi \in \hat{\mathbf{in}}(me(x))$, and defined the **out** of a set as the union of the **out** of its members, then we certainly have

$$\hat{\mathbf{out}}(\chi) \subseteq \hat{\mathbf{in}}(me(y))$$

But now $me(y) = \beta$ in the case of an input prefix by the definition of $\mathcal{G}$, thus we obtain

$$\hat{\mathbf{out}}(\chi) \subseteq \hat{\mathbf{in}}(\beta)$$

showing that $\hat{\mathbf{in}}$ and $\hat{\mathbf{out}}$ satisfy the same constraints as ρ and κ in this case.

[Subcase 1b: $x \notin \mathbf{bi}(P)$]
Suppose that $\{\chi\} \subseteq \hat{\mathbf{in}}(me(x))$, since x is not bound by input, and the fact that we adopted the convention that $\mathbf{in}_P(x) = x$ in this case, we have that $\chi \subseteq me(x)$ by definition of $\mathbf{in}_P$. By line 8 in our algorithm we have that

$$\mathbf{out}(x) \subseteq \mathbf{in}(y)$$

Using the marker environment in the same way as we did to obtain equation 3 we can infer that

$$\hat{\mathbf{out}}(me(x)) \subseteq \hat{\mathbf{in}}(me(y))$$

By definition of $\mathcal{G}$ we have $me(y) = \beta$. Since x is not bound by input, and the fact that we adopted the convention that $\mathbf{in}_P(x) = x$ in this case, we have that $\chi = me(x)$ by definition of $\mathbf{in}_P$ and our supposition. Substituting into the above we get $\hat{\mathbf{out}}(\chi) \subseteq \hat{\mathbf{in}}(\beta)$ as desired.

(Case 2: Output Prefix $\bar{x}y$) In this case the constraint set generated by $\mathcal{G}$ is

$$\{(\{\{\chi\} \subseteq \rho(me(x))\}) \Rightarrow \rho(me(y)) \subseteq \kappa(\chi) \mid \chi \in \mathcal{C}\}$$

We now subdivide into the four subcases accounting for the possible ways in which x and y can be bound:

[Subcase 2a: $\{x, y\} \subseteq \mathbf{bi}(P)$] Suppose that $\{\chi\} \subseteq \hat{\mathbf{in}}(me(x))$, by line 11 of our algorithm we have that $\mathbf{in}(y) \subseteq \mathbf{out}(\mathbf{in}(x))$, since we update the outsets of every $z \in \mathbf{in}(x)$ and defined the **out** of a set as the union of the **out** sets of its members. As in the above cases, we can infer that $\hat{\mathbf{in}}(me(y)) \subseteq \hat{\mathbf{out}}(\hat{\mathbf{in}}(me(x))$. Substituting χ for $\hat{\mathbf{in}}(me(x))$ by our supposition yields $\hat{\mathbf{in}}(me(y)) \subseteq \hat{\mathbf{out}}(\chi)$ as desired.

[2b $x \in \mathbf{bi}(P)$ **and** $y \notin \mathbf{bi}(P)$]
Suppose that $\{\chi\} \subseteq \hat{\mathbf{in}}(me(x))$, by line 13 of our algorithm we have that $y \in \mathbf{out}(\mathbf{in}(x))$, and thus (as above) that $me(y) \in \hat{\mathbf{out}}(\hat{\mathbf{in}}(me(x)))$. Since y is not bound by input we similarly have $\hat{\mathbf{in}}(me(y)) = me(y)$ by definition of $\mathbf{in}_P$. Sustituting χ for $\hat{\mathbf{in}}(me(x))$ by our supposition we thus get $\hat{\mathbf{in}}(me(y)) \subseteq \hat{\mathbf{out}}(\chi)$ as desired.

[2c $x \notin \mathbf{bi}(P)$ and $y \in \mathbf{bi}(P)$]
Suppose that $\{\chi\} \subseteq \hat{\mathbf{in}}(me(x))$, thus since x is not bound by input we have $\hat{\mathbf{in}}(x) = x$ as above. This implies that $\chi = me(x)$. By line 15 of our algorithm and our usual method from equation 3 have that $\hat{\mathbf{in}}(me(y)) \subseteq \hat{\mathbf{out}}(me(x))$. So $\hat{\mathbf{in}}(me(y)) \subseteq \hat{\mathbf{out}}(\chi)$.

[2d $x \notin \mathbf{bi}(P)$ and $y \notin \mathbf{bi}(P)$]
Suppose that $\{\chi\} \subseteq \hat{\mathbf{in}}(me(x))$, we again have that $\chi = me(x)$ as in the previous subcase, and by line 17 of our algorithm and our usual method we have that $me(y) \in \hat{\mathbf{out}}(me(x))$. Since y is not bound by input we also have that $\hat{\mathbf{in}}(me(y)) \subseteq \hat{\mathbf{out}}(me(x))$ as in subcase 2b. Thus, substituting χ as per our supposition, we get $\hat{\mathbf{in}}(me(y)) \subseteq \hat{\mathbf{out}}(\chi)$.

Thus we obtain that our solution $(\mathbf{in}, \mathbf{out})_P$, when composed appropriately with the marker environment generated by $\mathcal{G}[\![\hat{P}]\!]_{me}$ satisfies the constraints.$\square$

Proof of Lemma 9.4 We assume an appropriately α-converted process. Observe that the function pairs in our comparison are well typed, and thus can be compared using $\sqsubseteq$. For clarity, I use $\hat{\mathbf{in}}$ and $\hat{\mathbf{out}}$ as in the previous proof, as well as the following abbreviations:

$$\begin{aligned} \hat{\rho} &\stackrel{def}{=} (me \circ \rho \circ me^{-1}) \\ \hat{\kappa} &\stackrel{def}{=} (me \circ \kappa \circ me^{-1}) \end{aligned}$$

I also omit the P and $\hat{P}$ subscripts as they are now implicit. As in the previous proof (but in the opposite direction), we will now show that the function pair $(\hat{\rho}, \hat{\kappa})$ satisfies the inequalities implicit in our algorithm. We proceed by cases as before:

(Case 1: Input Prefix $x(y)$) By definition the following constraints are in $\mathcal{G}[\![\hat{P}]\!]_{me}$:

$$\{(\{\{\chi\} \subseteq \rho(me(x))\}) \Rightarrow \kappa(\chi) \subseteq \rho(\beta) \mid \chi \in \mathcal{C}\}$$

Where me maps y to β. We now consider the binding of x in separate subcases:

[Subcase 1a: $x \in \mathbf{bi}(P)$]
Consider an arbitrary $\chi \in \rho(me(x))$, by the definition of $\mathcal{G}$ we know that $\kappa(\chi) \subseteq \rho(\beta)$. Since this applies for all $\chi \in \rho(me(x))$, it also applies to the union, hence by substituting and generalizing κ to sets in the obvious way we get $\kappa(\rho(me(x)) \subseteq \rho(\beta)$. Using the marker environment as in the cases of the previous proof, and the fact that me maps β to y by the constraints we obtain

$$\hat{\kappa}(\hat{\rho}(x)) \subseteq \hat{\rho}(y)$$

But by line 6 of our algorithm, we have $\mathbf{out}(\mathbf{in}(x)) \subseteq \mathbf{in}(y)$. Hence $(\hat{\rho}, \hat{\kappa})$ satisfy the same inequality as $(\mathbf{in}, \mathbf{out})$ implying that

$$(\mathbf{in}, \mathbf{out}) \sqsubseteq (\hat{\rho}, \hat{\kappa})$$

Deconstructing $(\hat{\rho}, \hat{\kappa})$ and using composition with me we get

$$\begin{array}{rcl} (\mathbf{in}, \mathbf{out}) & \sqsubseteq & me \circ (\rho, \kappa) \circ me^{-1} \\ me^{-1} \circ (\mathbf{in}, \mathbf{out}) \circ me & \sqsubseteq & me^{-1} \circ me \circ (\rho, \kappa) \circ me^{-1} \circ me \\ (\hat{\mathbf{in}}, \hat{\mathbf{out}}) & \sqsubseteq & (\rho, \kappa) \end{array}$$

directly yielding the result for this case.

[Subcase 1b: $x \notin \mathbf{bi}(P)$]

Consider an arbitrary $\chi \in \rho(me(x))$, by the definition of $\mathcal{G}$ we know that $\kappa(\chi) \subseteq \rho(\beta)$. Since this applies for all $\chi \in \rho(me(x))$, it also applies to the union, hence by substituting and generalizing κ to sets in the obvious way we get $\kappa(\rho(me(x)) \subseteq \rho(\beta)$. Since x is not bound by input we have $me(x) \in \mathcal{C}$ by definition and thus $\rho(me(x)) = me(x)$ by the convention adopted for ρ. Thus our inequality becomes $\kappa(me(x)) \subseteq \rho(\beta)$. We use the marker environment as before to obtain $\hat{\kappa}(x) \subseteq \hat{\rho}(y)$, and since line 8 of our algorithm gives us $\mathbf{out}(x) \subseteq \mathbf{in}(y)$ we know that $(\mathbf{in}, \mathbf{out}) \subseteq (\hat{\rho}, \hat{\kappa})$. Composing with the marker environment as in the previous case yields the result.

(Case 2: Output Prefix $\bar{x}y$) By definition the following constraints are in $\mathcal{G}[\![\hat{P}]\!]_{me}$:

$$\{(\{\{\chi\} \subseteq \rho(me(x))\}) \Rightarrow \rho(me(y)) \subseteq \kappa(\chi) \mid \chi \in \mathcal{C}\}$$

Consider an arbitrary $\chi \in \rho(me(x))$, by the definition of $\mathcal{G}$ we know that $\rho(me(y)) \subseteq \kappa(\chi)$. Since this applies for all $\chi \in \rho(me(x))$, it also applies to their union under κ, hence by substituting and generalizing κ to sets in the obvious way we get

$$\rho(me(y)) \subseteq \kappa(\rho(me(x)) \tag{4}$$

We now subdivide into the four subcases accounting for the possible ways in which x and y can be bound:

[Subcase 2a: $\{x, y\} \subseteq \mathbf{bi}(P)$]

Marker environment composition makes inequality 4 become $\hat{\rho}(y) \subseteq \hat{\kappa}(\hat{\rho}(x))$. Line 11 of our algorithm gives us $\mathbf{in}(y) \subseteq \mathbf{out}(\mathbf{in}(x))$, and thus $\hat{\rho}$ and $\hat{\kappa}$ satisfy the inequalities implicit in the algorithm yielding the result through the usual manipulation of the marker environment.

[Subcase 2b: $x \in \mathbf{bi}(P)$ **and** $y \notin \mathbf{bi}(P)$]

Since y is not bound by input, $\rho(me(y)) = me(y)$ and our satisfied constraint becomes $\{me(y)\} \subseteq \kappa(\rho(me(x))$. Marker environment composition gives us $\{y\} \subseteq \hat{\kappa}(\hat{\rho}(x))$. Line 13 of our algorithm gives us $y \in \mathbf{out}(\mathbf{in}(x))$, and we have the desired correspondence between $(\hat{\rho}, \hat{\kappa})$ and $(\mathbf{in}, \mathbf{out})$.

[2c $x \notin \mathbf{bi}(P)$ **and** $y \in \mathbf{bi}(P)$ **]**

Since x is not bound by input inequality 4 becomes $\rho(me(y)) \subseteq \kappa(me(x))$, which can be turned to $\hat{\rho}(y) \subseteq \hat{\kappa}(x)$ with marker environment composition. Line 15 of our algorithm gives us $\mathbf{in}(y) \subseteq \mathbf{out}(x)$ and the desired relation is immediately yielded.

[2d $x \notin \mathbf{bi}(P)$ **and** $y \notin \mathbf{bi}(P)$ **]**

Since neither x nor y are bound by input, inequality 4 becomes $\{me(y)\} \subseteq \kappa(me(x))$. The usual marker environment composition makes this $\{y\} \subseteq \hat{\kappa}(x)$, and line 17 of our algorithm gives us the desired $\{y\} \subseteq \mathbf{out}(x)$ achieving the desired correspondence between the two solutions.

Thus we obtain that our solution $(\mathbf{in}, \mathbf{out})_P$, when composed appropriately with the marker environment generated by $\mathcal{G}[\![\hat{P}]\!]_{me}$ is at least as good as (ρ, κ).$\Box$