# DATAFLOW ANALYSIS OF THE Π-CALCULUS

*by*

*Sam B. Sanjabi*

School of Computer Science

McGill University, Montreal

June 2004

A thesis submitted to McGill University

in partial fulfillment of the requirements of the degree of

Master of Science

# Abstract

Static analysis [NNH99] is technique used to compute information about the run-time behaviour of a program prior to execution. Traditionally, it has been used in the context of optimizing compilers, but it has recently been applied to more formalized languages in order to develop provable policies that can be used to verify the security of networks. Best results are naturally achieved with the most precise information flow techniques, though complex systems impose feasibility constraints. Accuracy of results, particularly with respect to relative cost of computation is thus an important quality.

This thesis presents a series of dataflow analyses of the $\pi$-calculus, an extensively studied concurrent language that has been used to model and verify security protocols. Some of the presented analyses are equivalent to previous work done in the field, but the framework in which the analysis is done is new in that it immediately suggests an iterative implementation.

There are also analyses presented that improve on existing approaches in two ways. First, by fully treating the sequentiality of potential actions in a protocol, thereby improving the accuracy of previous approaches. Second, by considering the potential environment that a process could be running in, the computed results are correct independent of any context that the analyzed process may be in parallel composition with.

# Résumé

L'analyse statique [NNH99] est une technique qui permet à prédire de l'information à propos du comportement d'un programme avant l'exécution. Traditionnellement, elle a été utilisé durant l'optimisation des programmes durant la compilation, mais récemment cette mthode a été appliqué aux langages formelles pour formuler des politiques démontrable qui peuvent être utiliser pour vérifier la sécurité des réseau. Naturellement, les meilleurs résultats sont achevé avec les techniques les plus précis, donc l'exactitude des résultats, particulièrement à l'égard du prix relatif de l'exécution est une qualité importante à considérer.

Cette thèse présente une série d'analyses de l'écoulement de l'information dans le calcul-$\pi$, un langage concourant qui peut être utilisé pour modeler et vérifier les protocoles de sécurité. Certain des analyses présentés sont égales aux analyses qui existent déjà, mais la méthode avec lequel les analyses sont décris est nouvelle et suggère immdiatement des algorithmes itératives.

Des analyses qui améliorent les algorithmes éxistantes sont aussi inclus, et ces avancements sont fait de deux façons. Premièrement, en considérant la séquence des actions dans un protocole, améliorant la préçision de l'analyse. Deuxièmement, en considérant l'environnement potentiel dans lequel le procès pourrait exécuter, les résultats de l'analyse seront correcte sans dépendence sur le contexte avec lequel le programme est composé.

# Acknowledgements

First and foremost, my most heartfelt thanks must go to my thesis supervisor Clark Verbrugge. While we were both fairly new to the field, his constant encouragement and insistence on my solicitation of feedback through peer review and workshop attendance was absolutely vital to my timely completion of this thesis. Beyond that, professor Verbrugge is to be thanked for his valuable comments, and constant willingness to listen to my ideas.

I would also like to thank the FQRNT and NSERC for their partial financial support of this work.

My warm thanks also go to all of the members of the Sable Computing Laboratory at McGill that I have dealt with during my time here, it was a pleasure to work among such a fine group of people. Finally, I would like to thank my friends and family for their moral support and guidance over the past year.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Static program analysis is a technique often used in optimizing compilers to ascertain information about the run-time behaviour of a computer program prior to execution. Recently, static analysis techniques have been used in various other settings to compute information that would be useful beyond optimization. Notably, static analysis techniques can be used on parallel languages to ascertain properties of concurrent systems. One such language is the $\pi$-calculus, which models concurrency through name-passing semantics. This thesis develops a static analysis of the $\pi$-calculus that computes dataflow information through the modelled network. One notable application of such an analysis is the compile-time checking of various security properties of a network.

Traditionally, the security of networks has been enforced by the use of local technologies such as cryptography, and has been analyzed and validated through inspection by experts. With the advent of widespread heterogeneous public networks, information security has become a critical global issue rather than a locally desired one. Consequently, traditional security paradigms have become antiquated: while cryptography is an essential tool in network security, its very presence is not sufficient to

provide desired security guarantees, as its misuse can easily lead to insecure systems.

For instance, consider a protocol between two agents $A$ and $B$ in which a message $M$ is sent from $A$ to $B$ by encrypting it with a shared key $K_{AB}$. Let $\{M\}_{AB}$ represent such an encrypted message under the assumption that it is impossible to recover the message without knowledge of the key. Even with perfect encryption technologies used as tools, it is possible to write insecure protocols: if $A$ were to simply send the key $K$ to $B$ over a public channel, then an eavesdropper could easily recover the secret message by intercepting $K$.

More subtly, consider a system in which some number of agents communicate with a trusted server $S$ in order to establish shared keys, which could then be used to send encrypted messages. Letting $A$ and $B$ range over the agents, consider the following protocol (taken from [AG98]) in which $A$ sends a message $M$ to $B$ encrypted with a shared key $K_{AB}$ established with the help of $S$:

$$\begin{array}{lll} \text{Message 1} & A \rightarrow S: & A, \{B, K_{AB}\}_{AS} \quad \text{on } c_S \\ \text{Message 2} & S \rightarrow B: & \{A, K_{AB}\}_{SB} \quad \text{on } c_B \\ \text{Message 3} & A \rightarrow B: & A, \{M\}_{AB} \quad \text{on } c_B \end{array}$$

$K_{XY}$ represents a shared key between agents $X$ and $Y$, $c_X$ represents a public channel on which agent $X$ is awaiting input, and each line of the protocol of the form

$$\text{Message i} \quad X \rightarrow Y: \quad M \quad \text{on } c_Y$$

represents $X$ sending message $M$ to $Y$ on channel $c_Y$. The two agent version of this protocol is the well known Wide-Mouthed Frog protocol, and it is shown diagrammatically in figure 1. Note that it is initially assumed that the keys $K_{AS}$ and $K_{SB}$ are already established. While it is not immediately obvious, this protocol is seriously flawed with respect to authentication (see [AG98]): specifically, it is susceptible to a replay attack in which an adversary $E$ could convince $B$ that he is $A$ by intercepting and replaying $A$'s messages.

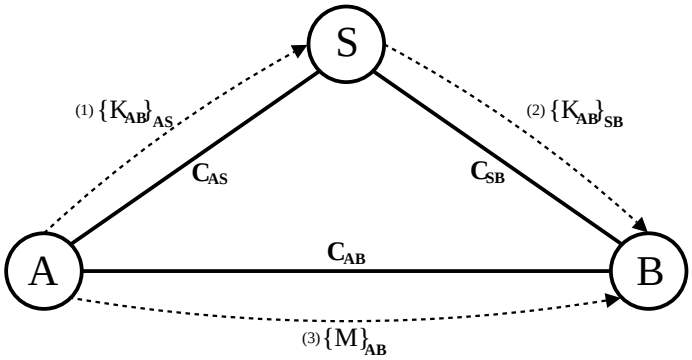



Figure 1.1: The Wide-Mouthed Frog Protocol

In addition to the ease with which flaws can trickle into systems, it is impractical to expect that each system can be correctly inspected by a knowledgeable party, as the number of networks requiring such inspection is astronomical, and the possibility of human error is always present. Thus the necessity of providing tools to automatically detect (and possibly repair) security flaws in systems is evident. It should also be noted that a reasonable solution to the problem should preferably not involve checking any heuristic conditions, as these could in turn prove ineffective. Rather, it would be better to use formal methods to precisely verify mathematically *provable* properties of systems, where the systems and properties themselves are also formally specified.

This thesis focuses on a framework in which such an analysis could be performed. Namely, a static analysis of the flow of data through concurrent systems as specified by Milner, Parrow and Walker's $\pi$-calculus [MPW92] is presented. This technique has been shown to be consistent with currently known analyses of the $\pi$-calculus, and has also led to algorithms that produce results that improve on the accuracy of previous approaches. The abstract depiction of dataflow presented here provides meaningful insight into the behaviour of processes, and makes further improvements in accuracy very clear. For the sake of preserving this clarity, the most efficient

algorithms for doing such analyses are not always discussed, but the appropriate references where efficiency is considered are cited in chapter 8. Efficiency is at best a tertiary consideration of this work, the focus here is on developing a technique in which flow analyses in concurrent systems (in the context of the $\pi$-calculus) can be visualized and improved in terms of accuracy.

## 1.1 Roadmap

Chapter 2 provides an overview of some of the work done in the field of formal security verification, this includes work that is quite distant in its scope from the particular niche dealt with by the rest of the document. However, it provides the broad context in which the work presented in this thesis can be placed. Chapter 3 presents the technical background on the $\pi$-calculus required to understand the rest of the work. Chapters 4, 5, and 6 develop a series of dataflow analyses of the $\pi$-calculus. The presentation of each such analysis is comprised of three parts:

1. A simplified representation of each $\pi$-calculus process that underlies the analysis itself

2. A description of the analysis algorithm that exploits the representation in order to compute some correct information about the process

3. A correctness proof

The sections detailing the proofs of correctness can safely be left out of a first reading without hindering the reader's comprehension of later chapters. Chapter 4 develops a simple flow insensitive analysis of the $\pi$-calculus that is a simplified version of previous work [BDNN01b] adapted into the iterative framework used to express subsequent analyses. Chapter 5 extends the analysis to produce some context independent results, and integrates a partial treatment of the sequential behaviour of

processes from [BDNN01b] into the framework, effectively providing a generalized version of the analysis presented there. Chapter 6 further extends the analysis to provide a full treatment of the sequentiality of processes, and is fully sensitive to the control flow of the process. The latter analysis is the most accurate analysis of the $\pi$-calculus known. Chapter 7 provides examples of the analyses of the previous chapters, and discusses how they can be used to statically check various security properties of programs. Lastly, chapter 8 discusses some of the limitations of the analyses, and provides extensions that could be integrated into the framework in the future in order to address them.

# Chapter 2

# Related Work

This chapter provides an overview of work done in the field of formal security verification. We begin with an enumeration of the formal languages used to express both systems and security properties, and then proceed with a list of the techniques that can be used to analyze these systems. An excellent survey of the work done in this area up to the end of 2002 is provided in [SM03].

## 2.1 System Description

Any security analysis of a system must be done with respect to a language in which that system can be expressed, thus we begin with a brief description of several formalisms suited for expressing sequential programs, concurrent systems, mobility, and cryptographic protocols.

Concurrency is often formally studied in the context of the $\pi$-calculus [MPW92], a process algebra introduced by Milner, Parrow and Walker. It provides a framework for the study of parallelism in a formal setting, and has been used extensively. The language is composed of building *processes* which represent agents that are able to

communicate by sending elements from a set of *names* over channels that are elements of the same set, i.e., there is no distinction in the language between data and the medium over which it is sent. It is this property that embodies the concept of *mobile computation*: this identification allows network structure to dynamically evolve, and is precisely what give the calculus its expressive power.

There is a lack of consensus on what the "standard" $\pi$-calculus is, but a formal definition of the calculus used as the basis of the analyses in this thesis is presented in the next chapter.

Other formalisms modelling concurrent systems include Hoare's CSP [Hoa85] and Milner's CCS [Mil89] which define calculi of communicating processes without the mobility aspect of the $\pi$-calculus, The Linda [CG89] language models parallelism as a globally shared space from which data can be synchronously or asynchronously read and written by agents. The Chemical Abstract Machine (CHAM) [BB92] defines machine states in a chemical model, i.e., a state is a "solution" (context) where floating "molecules" (processes) can "interact" (communicate) according to "reaction rules" (operational semantics).

While parallel languages such as the $\pi$-calculus could in principle be used to describe cryptographic protocols such as the one described in chapter 1, it can be more convenient to assume the existence of basic cryptographic operations with which to write protocols. The primary advantage of this approach is that it shifts the focus of security analysis away from the particular cryptographic operations available, and towards the actual usage of the assumed operations within a larger system. Abadi and Gordon's spi-calculus [AG99, AG98] achieves this by extending the $\pi$-calculus with (assumed *perfect*) primitives that allow for the encryption and decryption of transmitted data. Any security property that has been proven for a system expressed in the spi-calculus can be weakened to apply to the same system with actual encryption and decryption operations taking the place of the ideal ones.

A standard communication-based model is not the only object of study when confronting the issues pertinent to the today's global computational landscape. Not only do individual processes communicate with one another, they also have the capability to do so within different computational sub-networks. This version of *mobile computing* is modelled by Cardelli and Gordon's ambient calculus [CG98]. The intention is to model the capability of computational systems (such as a laptop) to move between networks with different properties, or equivalently to move about within a heterogeneous network. This model of *mobile computing* can be contrasted with the mobile computation of the $\pi$-calculus.

The ambient calculus as described in [CG98] is comprised of two components: the pure ambient component which describes *ambients* which model the localization of processes and how processes can move from ambient to ambient, and the communication component which describes how processes can communicate within an ambient. The latter is effectively equivalent to the $\pi$-calculus and is therefore Turing complete, but the purely mobile subset of the calculus is also Turing complete [Zim00].

Various other formalisms exist to study different computational models. Sequential programs can be expressed by purely functional calculi such as the untyped $\lambda$-calculus [Bar84], typed languages such as PCF [Mit96], and imperative languages such as Idealized Algol (IA). In addition to process algebraic approaches, there are also ways in which sequents in some formal logic can be used to express and analyze systems. As remarked by Schmatikov and Hughes [HS02], while it is relatively easy to express a system in a process algebra, it is harder to formulate information hiding properties based on them. Conversely, it is quite easy to express system properties with modal logics such as [SS99, FHMV95, ABN89] but much more difficult to express the systems themselves. Schmatikov and Hughes' work [HS02] attempts to rectify this by bridging the gap between logics and algebrae through the introduction of an interface layer built on the theory of function views.

## 2.2 System Analysis

Once an appropriate formalism in which to describe the system to be analyzed has been chosen, there are numerous techniques which can be used in order to infer the behaviour of that system. This section describes a number of such techniques.

### 2.2.1 Static Analysis

Static program analysis [NNH99] 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 programming language compiler. In such cases, the properties ascertained may have been the detection of redundant memory operations [CX02], compile-time type determination in languages with references [PR94], or virtual function resolution in object oriented languages [PR96, Sun99, SHR+00]. Recent work on language-based security has seen static analysis techniques applied to various process formalisms in order to compute information about the possible flow of data through a system at run-time. Such analyses can then be used to formulate security policies that can be used as safety criteria in networking applications. Static analysis ascertains properties of a program that 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.

The analysis is "static" in the sense that it computes information about programs based only on their syntax, the results will therefore apply to any possible execution of the program regardless of any variance in the inputs. The results can hence be used to accept or reject programs based on some criteria, i.e., they can be viewed as a type checking algorithm that restrict syntactically valid programs in some language from being considered "valid" according to some rule.

There are generally two ways to define a static analysis: expressing the analysis

as a type system, or as an abstract interpretation. These approaches are described below.

### Type Constraints

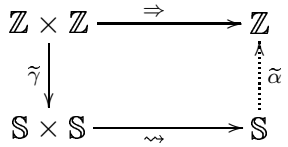
As noted above, a static analysis can be viewed as a type-checking algorithm in that it may be used to compute some information about the program (in traditional type-checkers, this would be referred to as the "type" of the program in question), and rejecting some programs based on this information (possibly because the type-checking failed, or the program's type was not the expected one).

Debbabi *et al.* [DDMM03] present a security analysis of protocols in which the sought type is an actual attack scenario on the input protocol. Thus if the protocol specification successfully type checks, this implies that a security flaw has been found. Aiken *et al.* present a line of research in [AW92, AW93, AFFS98a, AFF99] in which they build a framework for doing static analysis using type constraints. This theory is ultimately used to do points-to analysis on an imperative language with references [AFF97, AFF00], detecting races in RLL programs [AFS00], and exception tracking in ML [AFFC98]. This type theory also led to the creation of a toolkit for implementing constraint-based analyses [AFFS98b]. Volpano *et al.* [VSI96] present a type system specific to secure flow analysis. Bodei *et al.* present a series of work in [BDNN98,BDNN01b,BDPZ03] which deals with data flow analysis in the $\pi$-calculus by establishing the existence of a solution through its characterization in a flow logic, then by proving its correctness with a standard subject reduction result with respect to the operational semantics of the calculus. An actual solution that satisfies this characterization is then computed by solving a set of statically checkable type constraints. This work is discussed in greater detail in appendix A as it is used as a point of comparison for the analyses presented in this thesis. Quite similar techniques are

used in [BDNN01a] and [NNHJ99] for static analyses on the spi-calculus and the ambient calculus respectively. Finally Oxhöj, Palsberg, and Schwartzbach [OPS92,PS95] have also done work on safety analysis and type inference.

### Abstract Interpretation

Abstract interpretation [CC77] of programs is the process of describing the computations of a program (which can be viewed as functions over some set) in some abstract universe of objects, such that the results of the corresponding abstract execution gives some information about the actual computations. An intuitive example from arithmetic (which is borrowed from [CC77]) is the rule of signs. Integer arithmetic operations denote computations over the set of integers, i.e., the operations $* : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ and $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ can be viewed as functions over $\mathbb{Z}$. Thus the "program" $-213 * 11$ can be said to "execute" (or evaluate) to yield the result $-2343$. However, rather than viewing these operations over integers, we may abstractly view them as functions over the much smaller set $\mathbb{S} = \{(+), (-), (\perp)\}$ with the standard domain ordering $\preceq$ such that $(+) \succeq (\perp) \preceq (-)$. In other words, we *abstract* (or represent) integers by their sign, and ignore their value. Thus the execution of the above "program" $-213 * 11 \Rightarrow -2343$ would be interpreted in the abstract universe as $(-)\widetilde{*}(+) \rightsquigarrow (-)$ (using the abstract operations $\widetilde{*} : \mathbb{S} \times \mathbb{S} \to \mathbb{S}$ and $\widetilde{+} : \mathbb{S} \times \mathbb{S} \to \mathbb{S}$), thereby proving that the program $-213 * 11$ evaluates to a negative number without actually having to evaluate it. Thus, in a sense, we can say that this interpretation has yielded a static analysis of the language of the arithmetic operators $+$ and $*$ over the integers $\mathbb{Z}$ (modulo any overflow that could occur in practice). Letting $\Rightarrow$ and $\rightsquigarrow$ represent the concrete and abstract execution operators respectively, and defining $\widetilde{\gamma}$ and $\widetilde{\alpha}$ as the abstraction (i.e., taking the sign of) and concretion operations respectively, this process can be illustrated by the following diagram:




Observe that this abstraction has provided a summary of some facet of the actual execution of the program, but that the summary itself is incomplete. This is why the concretion arrow above is dotted: concretion is a hypothetical operation, information is lost during the abstraction process that cannot be regained. For example, consider the program $-213 + 11 \Rightarrow -202$, our abstract interpretation would execute $(-) \mathbin{\widetilde{+}} (+) \rightsquigarrow (\bot)$ which would not yield any information about the sign of $-213+11$. This illustrates the fact that abstract interpretation allows us to compute correct information about the run-time behaviour of a program in the sense that the computed information does not yield any false results. However, the abstract results are only a conservative approximations of the concrete execution of the program, i.e., they are ideal for expressing static analysis results.

Nielson *et al.* [NHN03] define an abstract interpretation of mobile ambients in a set-constraint based formalism. Malacaria and Hankin [MH98a] present an abstract interpretation of sequential programs based on game semantics [AM98, BDER97] which is used to compute an information flow analysis leading to a provable security property in Idealized Algol [MH98b, MH99]. Feret develops an abstract interpretation of a variant of the $\pi$-calculus which tracks multiple occurrences of subprocesses [Fer01], and is also used to prove a confidentiality property [Fer00].

### 2.2.2 Semantic Analysis

In contrast to the static techniques discussed above, there has also been work done on a different approach to ensure language-based security. Namely, the approach involves

the design of a language such that a desired security policy cannot be violated by programs written in the language. While this thesis focuses only on static techniques, a mention of this alternative approach is included for context.[1]

Often, the security property that such a language imposes will be some variant of "non-interference" which essentially means that *a variation of confidential input does not cause a variation of public output*. This notion can be rigorously formalized using a language's semantics by assuming that the state $s$ of a program point at some level involves the listing of the high (secret) and low (public) variables at that point, i.e., $s = (s_h, s_l)$. Letting $S$ denote the set of states, we can view the semantics $[\![C]\!]$ of a program $C$ as a function $[\![C]\!] : S \to S_\perp$ (where $S_\perp = S \cup \{\perp\}$) that maps an input state $s \in S$ either to an output state $[\![C]\!]s \in S$, or to $\perp$ if $C$ fails to terminate. Variation of secret input can be described as an equivalence relation $=_L$ on states, which says that two states are equal whenever they agree on public values (i.e., $s =_L s'$ iff $s_l = s'_l$). Then the observational power of a potential attacker can be modelled by a relation $\approx_L$ on behaviours such that two behaviours are related by $\approx_L$ iff they are indistinguishable to an attacker. The choice of $\approx_L$ depends on the specific security property desired (and thus on the attacker model). Thus non-interference can be formalized as follows: $C$ is secure iff

$$\forall s_1, s_2 \in S. s_1 =_L s_2 \Longrightarrow [\![C]\!]s_1 \approx_L [\![C]\!]s_2$$

Examples of this approach include the SLam calculus [HR98], an extension of the $\lambda$-calculus that enforces secrecy and integrity properties. Agat [Aga00] discusses how imperative programs could be transformed so that information leaks due to timing channels could be closed. Abadi and Gordon [AG99, AG97] prove security properties based on process equivalences in the spi-calculus. Finally, [CGG02] provides an extension of the $\pi$-calculus using sorts (or "groups") that ensure a secrecy property

[1] Much of this brief discussion is taken from [SM03]

through sort checking.

While semantic analyses can formalize extremely accurate properties of a program, they are often based on equivalences that aren't decidable in general. Thus the truly interesting area of research is the connection between the static and semantic approaches, i.e., getting results that are as accurate as possible while maintaining practicality. This thesis focuses on that connection in the context of concurrent networks.

# Chapter 3

# Background: The $\Pi$-Calculus

This section presents the background required to comprehend the rest of the thesis. Specifically, a complete description of the formal semantics of the variant of the monadic $\pi$-calculus studied in the following sections is presented.

## 3.1 Syntax

Assuming a countably infinite set of names $\mathcal{N} = \{a, b, \ldots, x, y, \ldots\}$, and a distinguished element $\tau$ such that $\mathcal{N} \cap \{\tau\} = \emptyset$. A *process* in the $\pi$-calculus is a term built from the following syntax:

$$\begin{aligned} \pi &::= \tau \mid \overline{x}\langle y\rangle \mid x(y) \mid [x = y] \\ P &::= \mathbf{0} \mid \pi.P \mid P + P \mid P \| P \mid (\nu x)P \mid\, !P \end{aligned}$$

The production $\pi$ defines the allowable *prefixes* (or actions) of a process $P$. Given a process $P$, the set of *output prefixes* (i.e., prefixes of the form $\overline{x}\langle y\rangle$) of $P$ is denoted $\mathcal{O}_P$, and the set of its *input prefixes* (i.e., prefixes of the form $x(y)$) is denoted $\mathcal{I}_P$. The set of *match prefixes* (i.e., prefixes of the form $[x = y]$) is denoted $\mathcal{M}_P$. The set

of *observable prefixes* of $P$ is defined as $\mathcal{X}_P = \mathcal{O}_P \cup \mathcal{I}_P$. The name playing the role of the communication channel in an observable prefix $\pi$ is called the *channel* of $\pi$ (denoted $\mathrm{cha}[\pi]$), and the *datum* of $\pi$ (denoted $\mathrm{dat}[\pi]$) is similarly defined. Formally:
**Definition 3.1.1.**

$$\mathrm{cha}[x(y)] \triangleq x \qquad \mathrm{cha}[\overline{x}\langle y\rangle] \triangleq x$$
$$\mathrm{dat}[x(y)] \triangleq y \qquad \mathrm{dat}[\overline{x}\langle y\rangle] \triangleq y$$

The functions $\mathrm{cha}[\cdot]$ and $\mathrm{dat}[\cdot]$ are not defined on any other inputs. Lastly, the set of *all* prefixes of a process (observable prefixes and matches, excluding $\tau$) is denoted by $\Pi_P = \mathcal{X}_P \cup \mathcal{M}_P$.

The trailing $\mathbf{0}$ is hereafter omitted from any sequence of prefixes, i.e., $\pi$ is written rather than $\pi.\mathbf{0}$ for any prefix $\pi$. Input prefixes and restrictions bind names, e.g., the name $y$ in the process $x(y).P$ is *bound* in (or scoped to) $P$. Given a process $P$, the set of *names* $\mathrm{n}(P)$, and the set of *bound names* $\mathrm{bn}(P)$ are defined in the obvious way. The set of *free names* ($\mathrm{fn}(\cdot)$) of $P$ is defined as $\mathrm{fn}(P) = \mathrm{n}(P) \setminus \mathrm{bn}(P)$. A process $P$ is *closed* if $\mathrm{fn}(P) = \emptyset$, and *open* otherwise. Furthermore, the set $\beta(P)$ is defined as the set of names occurring as the bound parameter in an input prefix in $P$, and its complement (with respect to $P$) is denoted by $\chi(P) = \mathrm{n}(P) \setminus \beta(P)$.

Finally, a notion of substitution on processes is defined as follows: given a process $P$, the process $P\{x \mapsto y\}$ is defined as the same process as $P$ with every free occurrence of the name $x$ replaced by the name $y$ (under the assumption that $y \notin \mathrm{fn}(P)$).

## 3.2 Semantics

This section begins by informally introducing the meanings of the process constructors defined in the syntax above. The prefixes $x(y)$, $\overline{x}\langle y\rangle$, $\tau$ represent a process attempting synchronous input, output, and an unobservable (silent) action respectively. The match prefix $[x = y]$ models an idle process unless the names $x$ and $y$ are identical, in

| Construct | Name | Description |
|---|---|---|
| $\mathbf{0}$ | NIL | An idle process, a process that can take no actions |
| $x(y).P$ | INPUT | Binds the name $y$ in $P$, waits until it synchronously receives input $m$ on channel $x$, and then behaves as $P\{y \mapsto m\}$ |
| $\overline{x}\langle y\rangle.P$ | OUTPUT | Waits until it can synchronously send output $y$ on channel $x$ and then behaves as $P$ |
| $\tau.P$ | SILENT | Takes one step to do action $\tau$ at any point and then behaves as $P$ |
| $[x = y].P$ | MATCH | Behaves as $P$ if the name $x$ is the same as the name $y$, and behaves as $\mathbf{0}$ otherwise |
| $P_1 + P_2$ | CHOICE | Non-deterministically behaves as either $P_1$ or $P_2$ but not both |
| $P_1 \| P_2$ | COMPOSITION | Behaves as $P_1$ and $P_2$ executing in parallel, allowing them to communicate synchronously |
| $(\nu x)P$ | RESTRICTION | Binds the name $x$ in $P$, generates a new name $x$ to be used in the scope of the process $P$ |
| $!P$ | REPLICATION | Behaves as an infinite number of copies of $P$ in parallel composition (i.e., as $P \| P \| P \| \ldots$) |

Table 3.1: Informal meanings for the various syntactic constructs of the $\pi$-calculus.

which case the process is allowed to execute. The nil process $\mathbf{0}$ represents an inactive process. The + operator forces a non-deterministic choice between its operands, while the $\|$ operator places two processes in parallel composition with one another allowing them to execute simultaneously. The restriction operator $\nu$ creates new names, and the replication operator ! models the behaviour of its operand placed in parallel composition with infinite copies of itself. These informal behaviours are summarized in table 3.1.

These descriptions are formalized by a trio of relations on processes defined through precise logical inference rules on the structure of processes. These relations are defined formally in the following sections.

### 3.2.1 Structural Congruence

First, it is noted that $\alpha$-equivalence ($=_\alpha$) is defined as in [BDPZ03] by assuming that names are "stable", i.e., that for each name $a \in \mathcal{N}$ there is a canonical representative $\lfloor a \rfloor \in \lfloor \mathcal{N} \rfloor$. Two names are said to be $\alpha$-convertible only if they have the same canonical names, although hereafter we shall just write $a$ for $\lfloor a \rfloor$. In other words, if the set $\mathcal{N}$ is partitioned into countably infinite subsets $\{a, a', a_0, \ldots\}$, $\{b, b', b_0, \ldots\}$, $\ldots$, then two names can only be $\alpha$-converted if they belong to the same subset. This statement may be a technicality, but it is worth mentioning in order to statically maintain the identity of names that may be lost through unrestrained $\alpha$-conversions. The $\alpha$-equivalence relation $=_\alpha$ is necessary in order to rename the bound names of a process in order to avoid conflicts between them as the computation takes place.

Structural congruence on processes is then defined as the least relation on closed processes satisfying the rules in table 3.2. Most of these rules describe standard properties that processes should emulate (e.g., commutativity and associativity of processes), allowing us to identify processes that behave in the same way. We only point out the rule `(S Rep)` that allows copies of the process $P$ to be "peeled off" from the replicated process $!P$ as needed. This rule emulates the behaviour that $!P$ behaves as $P \| P \| \ldots$.

### 3.2.2 Operational Semantics

The complete operational semantics are given as a labelled transition system in table 3.3. The semantic rules define an early transition system defining a set of relations $\xrightarrow{\mu}$ between processes, with $\mu$ ranging over the set $\{ab, \overline{a}b, \overline{a}(b), \tau\}$ of *labels* for $\{a, b\} \subseteq \mathcal{N}$. A process $P$ is said to *reduce* to a process $Q$ if, using these inference rules, it can be proven that $P \xrightarrow{\tau} Q$. The binary predicate $\psi$ (defined at the top of table 3.3) is purely a notational convenience, and is only used as a condition for the

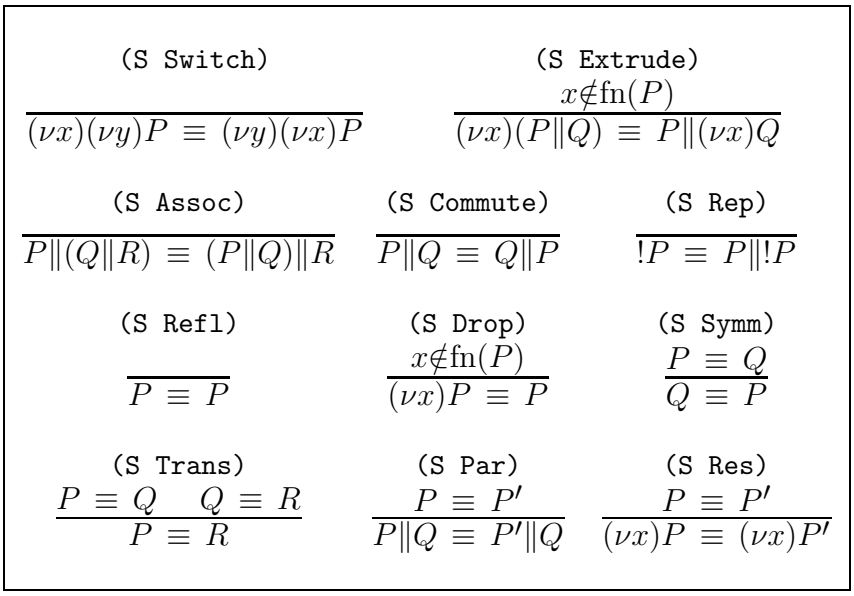



Table 3.2: Structural Congruence for the Π-Calculus

rule (R Par), note that the functions $\mathrm{bn}(\cdot)$ and $\mathrm{fn}(\cdot)$ are extended to labels in the obvious way.

An interesting property of the $\pi$-calculus that is relevant to dataflow analysis is scope extrusion. It refers to the ability of a process to send its bound names outside of its own scope, and is a consequence of the (R Close) rule. Consider the parallel composition $P \| Q$ with $P \triangleq (\nu a)\overline{x}\langle a \rangle.P'$ and $Q \triangleq x(b).Q'$. The process $P$ has a private name $a$, and wishes to send it on channel $x$, and by the (R Open) and (R Out) rules it can be derived that $P \xrightarrow{\overline{x}(a)} P'$. Meanwhile, $Q$ is waiting to receive a value on the very same channel $x$, and by the (R In) rule, $Q \xrightarrow{xa} Q'\{b \mapsto a\}$ can be derived. Thus the (R Close) rule can be used to derive that $P \| Q \xrightarrow{\tau} (\nu a)(P' \| Q'\{b \mapsto a\})$ is derivable. Effectively, the name $a$ which was bound in $P$ has been sent to another process, and its scope has been "extruded" to encompass both processes. This phenomenon is obviously important when trying to develop dataflow analyses that are correct independent of context.

$$\psi(\mu, P) \triangleq (bn(\mu) \ \cap \ fn(P) = \emptyset)$$

(R Tau)
$$\frac{}{\tau.P \xrightarrow{\tau} P}$$

(R In)
$$\frac{}{a(y).P \xrightarrow{ab} P\{b \mapsto y\}}$$

(R Out)
$$\frac{}{\overline{a}\langle b\rangle.P \xrightarrow{\overline{a}b} P}$$

(R Par)
$$\frac{P_0 \xrightarrow{\mu} Q_0 \qquad \psi(\mu,P_1)}{P_0 \| P_1 \xrightarrow{\mu} Q_0 \| P_1}$$

(R Sum)
$$\frac{P_0 \xrightarrow{\mu} Q_0}{P_0 + P_1 \xrightarrow{\mu} Q_0}$$

(R Close)
$$\frac{P_0 \xrightarrow{\overline{a}(b)} Q_0 \qquad P_1 \xrightarrow{ab} Q_1}{P_0 \| P_1 \xrightarrow{\tau} (\nu b)(Q_0 \| Q_1)}$$

(R Res)
$$\frac{P \xrightarrow{\mu} Q \qquad a \notin \mathrm{n}(\mu)}{(\nu a)P \xrightarrow{\mu} (\nu a)\ Q}$$

(R Open)
$$\frac{P \xrightarrow{\overline{a}b} Q \qquad b \neq a}{(\nu b)P \xrightarrow{\overline{a}(b)} Q}$$

(R Com)
$$\frac{P_0 \xrightarrow{\overline{a}b} Q_0 \qquad P_1 \xrightarrow{ab} Q_1}{P_0 \| P_1 \xrightarrow{\tau} Q_0 \| Q_1}$$

(R Match)
$$\frac{P \xrightarrow{\mu} Q}{[x{=}x].P \xrightarrow{\mu} Q}$$

(R Var)
$$\frac{P' \equiv P \qquad P \xrightarrow{\mu} Q \qquad Q \equiv Q'}{P' \xrightarrow{\mu} Q'}$$

Table 3.3: Operational Semantics of the Π-Calculus

# Chapter 4

# Static Dataflow Analysis

The $\pi$-calculus language described in the previous chapter can be used to model systems of processes executing in parallel. These can range in behaviour from modelling database servers and their potential clients, cryptographic protocols (perhaps with an extension to the language as in Abadi and Gordon's spi-calculus [AG99]), or even standard sequential operations such as arithmetic or propositional logic expressions. Regardless of the process in question, static analysis techniques can be applied to the formulation in order to efficiently compute an approximation of the process' potential behaviour. Such an approximation can be used to provide guarantees about the execution of the process (which can represent anything from a calculator to a network), such as the potential usage limits of the network, or the fact that the protocol that it represents respects a particular security policy.

As mentioned in chapter 2, static analyses can generally be formulated as a type system or as an abstract interpretation. This chapter begins the development of a new abstract interpretation for $\pi$-calculus processes. This abstraction is used to compute a dataflow analysis of the calculus, and the technique is shown to be able to model previous analyses [BDNN98, BDNN01b, BDPZ03]. Subsequent chapters work

to enhance the abstraction presented here in order to obtain increasingly refined process representations, and thus more accurate results than had previously been possible. The tradeoffs of such refinements are also discussed.

## 4.1 A First Abstraction of Processes

Analyzing $\pi$-calculus processes in general is a hard, and often undecidable problem. The semantic techniques described in section 2.2.2 are dynamic approaches that, while capable of providing extremely accurate results in theory, rely on the existence of an equivalence relation on the denotations of the systems under scrutiny. In the context of the $\pi$-calculus, such equivalences (bisimulations [Par81] or other observational equivalences) have not been shown to be decidable in general, and thus are questionable in terms of their practicality. Static analyses provide polynomial time algorithms because they safely approximate the behaviour of the process, and this approximation can then be used to efficiently check that a process conforms to a desired property.

In order to achieve such results, it is necessary to represent $\pi$-calculus processes in a simpler form that abstracts some of their behaviour, while still yielding correct results. The analogy to the discussion from section 2.2.2 is the representation of integers by their signs, this abstraction allows for the computation of partial information about an arithmetic expression (namely the sign of the final result), by using the efficient abstract operations $\widetilde{*}$ and $\widetilde{+}$ instead of the (relatively) expensive multiplication and addition operators on integers. In the same way, a $\pi$-calculus process can be abstracted to a simpler form that is amenable to worst-case polynomial time execution.

### 4.1.1 Flow Insensitive Representation

The only action (aside from the unobservable $\tau$ action) that a $\pi$-calculus process can take is that of synchronous communication between two processes composed in parallel with the $\|$ operator. A natural first abstraction on such a process is to view it as the set of actions that it could potentially perform *as a whole*, regardless of the sequencing or structure in which these actions could occur during the actual execution of the process. Specifically, we can view a $\pi$-calculus process $P$ as the set of observable prefixes (actions) that it contains, while ignoring the operators that sequence (through the . operator) or structure (with the $\|$, $+$, and !) these actions in particular ways. These ideas are formalized by the following definition

**Definition 4.1.1.** Given a $\pi$-calculus process $P$, and recalling the definition of $\mathcal{X}_P = \mathcal{O}_P \cup \mathcal{I}_P$ as the set of input and output prefixes of a process, the flow insensitive abstraction function $[\![P]\!]_\emptyset$ is defined as follows:

$$[\![P]\!]_\emptyset = \mathcal{X}_P$$

It is assumed that the bound names of $P$ are first $\alpha$-converted to avoid repetition.

Note that the match prefixes $\mathcal{M}_P$ of $P$ are also omitted from the denotation. The $\emptyset$ subscript indicates that no information about the preceding behaviour of each prefix is included in the representation. The representations in subsequent chapters, however, pair each prefix with subsets of the actions that precede them in order to reason about the sequential nature of the process. The analysis presented in this section therefore implicitly assumes that all tests (matches) succeed, and do not block any subsequent actions. This simplification is very conservative, but the analysis discussed in this section is useful nonetheless, as it is used as a building block for more accurate analyses.

The abstraction is flow insensitive because it ignores all sub-process structure, i.e., the following equations are satisfied for any names $a$,$b$,$c$,$d$:

$$[\![a(b) + \overline{c}\langle d\rangle]\!]_\emptyset = [\![a(b)\|\overline{c}\langle d\rangle]\!]_\emptyset = [\![a(b).\overline{c}\langle d\rangle]\!]_\emptyset = \{a(b), \overline{c}\langle d\rangle\}$$

An identical property applies for the abstractions of the replicated versions of the above processes. Briefly, given any process $P$, this abstraction does not differentiate between $P$ and $\widetilde{P}$ where $\widetilde{P}$ is the same as $P$ with all prefixing and choice operators replaced by $\|$ and replication operators moved to the top level. The unobservable $\tau$ action is excluded from the representation because it contributes nothing to dataflow within a process. The *abstraction* is context insensitive because the representation of the subprocess $P$ does not change if it is abstracted in another context. These properties directly imply the compositionality of the abstraction process stated here as a proposition (which could in fact be used as an alternative definition):

**Proposition 4.1.1.**

$$\begin{aligned} [\![P\|Q]\!]_\emptyset = [\![P+Q]\!]_\emptyset = &\quad [\![P]\!]_\emptyset \cup [\![Q]\!]_\emptyset && \forall P, Q \\ [\![(\nu x)P]\!]_\emptyset = [\![\tau.P]\!]_\emptyset = [\![!P]\!]_\emptyset = &\quad [\![P]\!]_\emptyset && \forall P\ \forall x \in \mathcal{N} \\ [\![\pi.P]\!]_\emptyset = &\quad \{\pi\} \cup [\![P]\!]_\emptyset && \forall P\ \forall \pi \in \mathcal{X}_P \end{aligned}$$

*Proof.* Immediate from definition 4.1.1. □

Definition 4.1.1 requires that $P$ be $\alpha$-converted to avoid name repetition. This is done in order to avoid needlessly identifying our analysis results for repeated names. Consider the process $P = \overline{x}\langle a\rangle + (\nu a)(\overline{x}\langle a\rangle)$ representing a process that has the capability of sending the name $a$ over the channel $x$ in one of two ways, however the name $a$ in the left sub-process is semantically distinct from the one in the right sub-process because the latter is bound in $(\nu a)(\overline{x}\langle a\rangle)$. In fact, $P$ has the capability of transmitting two different names over the channel $x$. Applying our abstraction to $P$ without first $\alpha$-converting the names appropriately would lose this distinction, i.e.,

the denotation of $P$ would be the singleton set $\{\overline{x}\langle a\rangle\}$. By $\alpha$-renaming the process to make the difference explicit, we can preserve that information in the abstract set, i.e., by renaming $P =_\alpha \overline{x}\langle a\rangle + (\nu a')(\overline{x}\langle a'\rangle)$ the abstraction would more accurately be computed as $[\![P]\!]_\emptyset = \{\overline{x}\langle a\rangle, \overline{x}\langle a'\rangle\}$. This assumption applies to all processes considered in this thesis and is formally stated here:

**Assumption 1.** Any analyzed process $P$ is pre-processed by $\alpha$-converting it to avoid syntactic repetition of bound names

Furthermore, it is assumed that each prefix in the process is labelled so as to distinguish between multiple occurrences of a prefix pair, and that a mapping between labels and occurrences of prefixes exists. It is these labels that are actually used in the representations presented in this thesis. In fact, any reference to a prefix actually refers to a label mapped to the *occurrence* of a prefix. For clarity, this additional step has been omitted from this treatment, and the examples presented avoid multiply occurring prefixes. However, this assumption must be kept in mind, and is stated formally here:

**Assumption 2.** Every prefix in a process is labelled, and there exists a mapping from labels to occurrences of prefixes. A reference to a prefix in the remainder of the document is actually a reference to a label denoting a particular occurrence of a prefix.

### 4.1.2 Abstract Execution

The development of an abstract representation of processes as sets of prefixes yields a simplified universe which is much easier to analyze than the universe of all possible processes. The next step is to develop a concept of execution in this universe that would glean some partial information about the potential behaviour of the original

$\pi$-calculus process. In order to do so, it is important to formally characterize the information that one wishes to compute.

Given that the only dataflow action that a $\pi$-calculus process can take is the transmission of names over channels, it is quite natural that the final solution of the abstract execution function include this information. Specifically, given the denotation of a $\pi$-calculus process $[\![P]\!]_\emptyset$, the abstract execution should compute a function $K$ taking each name $n \in \mathrm{n}(P)$ to a superset of the names that could be transmitted over $n$. In other words, if it is possible for $P$ to perform an action $P \xrightarrow{\overline{n}x} P'$ for some name $x$ (i.e., if $P$ can send $x$ over $n$), the final solution $K$ of the abstract execution should include $x \in K(n)$. While this notion is necessary for a full treatment of dataflow, it is insufficient, as the transmitted name $x$ could itself be bound by input (i.e., $x \in \beta(P)$). In this case, the name $x$ could actually be substituted for other names during execution (because $x$ acts as a datum in some input prefix), and thus these names should also be kept track of by the analysis. The set of names that a name $x$ could assume during execution is denoted by $R(x)$, and if it is possible for a process $P$ to transmit $x$ over a channel $n$ as above, then we must ensure that the final solution of the analysis includes the entirety of $R(x)$ in $K(x)$. Using like reasoning, we formally develop the characterization of our analysis solution below.

### A Lattice Of Solutions

For a given process $P$, the solutions computed by the abstract execution function will consist of pairs of functions $(R, K)$, where $R : \mathcal{N} \to 2^{\mathcal{N}}$ will yield the set of names that a name could assume during execution (where names not in $\beta(P)$ are mapped to themselves), and $K : \mathcal{N} \to 2^{\mathcal{N}}$ will yield the set of names that could directly be transmitted over the given name acting as a channel in an output prefix. Any names not involved in the process in question will be taken to bottom (i.e., the empty set)

by these functions.

It is a standard result that, given any countable set $S$, the powerset of $S$ (denoted $2^S$) forms a complete lattice $L$ in which inclusion ($\subseteq$) is the partial ordering ($\sqsubseteq_L$) in $L$, the union ($\cup$) and intersection ($\cap$) operations are the least upper bound ($\sqcup_L$) and greatest lower bound ($\sqcap_L$) operations in $L$ respectively, and $\emptyset$ and $S$ are the bottom ($\bot_L$) and top ($\top_L$) elements of $L$ respectively. Furthermore, it is known that the total function space between a countable set $S$ and a lattice $L$ also forms a complete lattice under the standard pointwise operations:

**Fact 4.1.1.** Given a countable set S and a lattice $(L, \sqsubseteq_L, \sqcup_L, \sqcap_L, \bot_L, \top_L)$, let $[S \Rightarrow L]$ denote the set of all total functions from $S$ to $L$. Given any $f, g \in [S \Rightarrow L]$, define the following:

$$\begin{aligned} f \sqsubseteq g &\Leftrightarrow \forall x \in S : f(x) \sqsubseteq_L g(x) \\ (f \sqcup g)(x) &= (f(x) \sqcup_L g(x)) \\ (f \sqcap g)(x) &= (f(x) \sqcap_L g(x)) \\ \bot(x) &= \bot_L \\ \top(x) &= \top_L \end{aligned}$$

for any $x \in S$, then $([S \Rightarrow L], \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ also forms a complete lattice.

It then trivially follows that the set $[\mathcal{N} \Rightarrow 2^{\mathcal{N}}]$ forms a complete lattice. The bottom element is the function that takes every input to the empty set, and shall be denoted $\bot_K$ as it will act as the bottom element for the $K$ component of the analysis solution.

Now, consider any subset $I \subseteq S$, and take any function $\bot_S^I \in [S \Rightarrow 2^S]$ that takes each element in $I$ to the singleton set composed of itself, i.e., $\bot_S^I(i) = \{i\}$ for each $i \in I$, and takes all other elements of $S$ to $\emptyset$. The subset of $[S \Rightarrow 2^S]$ composed of functions greater than $\bot_I^S$ also forms a complete lattice for any $I$:

**Proposition 4.1.2.** *Given a countable set $S$, its induced function space lattice $[S \Rightarrow 2^S]$, and given any subset $I \subseteq S$, if the function $\bot_S^I$ is defined as*

$$\bot_S^I(x) = \begin{cases} \{x\} & , \text{ if } x \in I \\ \emptyset & , \text{ otherwise} \end{cases}$$

*for any element $x \in S$, then the set $\mathcal{L}_S^I = \{f \in [S \Rightarrow 2^S] \mid \bot_S^I \sqsubseteq f\}$ forms a complete sub-lattice of $[S \Rightarrow 2^S]$ with the same partial ordering, least upper bound and greatest lower bound operations, and top element, and with bottom element $\bot_S^I$.*

*Proof.* The relation $\sqsubseteq$ is trivially still a partial ordering on the subset $\mathcal{L}_S^I$, and $\bot_S^I \sqsubseteq f$ for any $f \in \mathcal{L}_S^I$ by definition, and thus in fact is the bottom element of $\mathcal{L}_S^I$. Since $f \sqsubseteq \top$ for any $f \in [S \Rightarrow 2^S]$ then $\top$ is certainly greater than any function in $\mathcal{L}_S^I$, so the top element does not change. For any subset $F \subseteq \mathcal{L}_S^I$, $\bot_S^I$ is a lower bound of $F$ by definition of $\mathcal{L}_S^I$. Furthermore, $\sqcup F$ is an upper bound of $F$ by definition of $\sqcup$. Thus $\bot_S^I \sqsubseteq \sqcup F$ by transitivity of $\sqsubseteq$, and hence $\sqcup F \in \mathcal{L}_S^I$ by the definition of $\mathcal{L}_S^I$. Finally, $\bot_S^I$ is a lower bound on $F$ because it is the bottom element, and $\sqcap F$ exists in $[S \Rightarrow 2^S]$ because it is a lattice. Now suppose that it is not in $\mathcal{L}_S^I$, this implies that some $i \in I$ is mapped to $\emptyset$ by $\sqcap F$. Since the $\sqcap$ function takes the intersection of the images of each element of $S$ under $\sqcap F$, this implies that some function in $F$ also mapped $i$ to the empty set, contradicting the fact that $F \subseteq \mathcal{L}_S^I$, yielding the desired result that $\sqcap F \in \mathcal{L}_S^I$. □

This proposition implies that the set $\mathcal{L}_S^I \subseteq [\mathcal{N} \Rightarrow 2^{\mathcal{N}}]$ generated as above for some $I \subseteq \mathcal{N}$ forms a complete lattice denoted $\mathcal{L}_R^I$. The bottom element of this sub-lattice is denoted $\bot_R^I$, and shall act as the bottom element for the $R$ component of the analysis solution. As indicated by the notation, the subset $I$ taken will depend on the process in question: namely $I$ shall be defined as $I = \chi(P)$. This will have the effect of "initializing" the $R$ function to return $\{x\}$ for every name $x \in \chi(P)$[1].

[1]This captures the notion that every name not bound by input will at least represent itself during

Since complete lattices are closed under Cartesian product (again by ordering pairs pointwise), the set of all pairs of such functions $(R, K)$ also forms such a lattice, which shall be denoted by $\mathcal{L}_{\emptyset}^{P}$, and its bottom element is denoted $\bot_{\emptyset}^{P} = (\bot_{R}^{\chi(P)}, \bot_{K})$. This solution set is identical to that used by Bodei *et al.* in [BDNN98, BDNN01b] (see section A.1 in the appendix for an overview). The superscripts $P$ and $\chi(P)$ are hereafter omitted when the process in question is clear from the context.

### The Execution Function

As observed above, every prefix implies a potential communication: an output prefix implies that its datum should be included in the set of names that its channel could transmit, and an input prefix implies that the set of names transmitted over its channel and allowed by its filter should be included in the set of names that its datum could assume. This observation motivates the definition of the dataflow analysis as a function that iterates over sets of prefixes, while updating the solution functions in $\mathcal{L}_{\emptyset}$.

Prefixes can be formalized by pairs as follows:

- An output prefix $\overline{x}\langle y\rangle$ is formalized as a pair of the form $\langle x, y\rangle$, for $\{x, y\} \subseteq \mathcal{N}$, and the set of all such pairs $\langle \mathcal{N} \times \mathcal{N}\rangle$ is denoted by $\mathcal{O}$
- An input prefix $x(y)$ is formalized as a pair of the form $(x, y)$, for $\{x, y\} \subseteq \mathcal{N}$, and the set of all such pairs $(\mathcal{N} \times \mathcal{N})$ denoted by $\mathcal{I}$
- The set of all observable prefixes is then denoted by $\mathcal{X} = \mathcal{O} \cup \mathcal{I}$.

The shape of the brackets around the pairs are used only to indicate that the two isomorphic sets are kept distinguished. Viewed this way, the functions cha[·] and dat[·] can be seen as the first and second projections over these sets. The function

execution.

$\Omega : 2^{\mathcal{X}} \to \mathcal{L}_\emptyset \to \mathcal{L}_\emptyset$ is then defined as our abstract execution function. Intuitively, given a set of prefixes $S \subseteq \mathcal{X}$, this function can then be viewed (by currying $S$) as $\Omega_S : \mathcal{L}_\emptyset \to \mathcal{L}_\emptyset$ taking an initial pair of functions $(R, K) \in \mathcal{L}_\emptyset$ and returning a pair of functions that is updated by the information gleaned by iterating over the set $S$. Functions $f : \mathcal{L} \to \mathcal{L}$ over elements of lattices are extended to sets of elements in the obvious way, i.e., $\forall L \subseteq \mathcal{L} : f(L) = \bigsqcup_{x \in L} f(x)$ with $\sqcup$ denoting the join operation in the lattice $\mathcal{L}$. Similarly, the functions $\beta(\cdot)$ and $\chi(\cdot)$ previously defined on processes are extended to sets of prefixes, i.e., $\beta(S)$ is the set of names occurring as the datum in an input prefix etc.. For convenience, a special notation for singleton maps is defined as follows: for a countable set $X$ and a lattice $\mathcal{L}$, the map in $[X \Rightarrow \mathcal{L}]$ that takes an element $x \in X$ and returns an element $l \in \mathcal{L}$ and maps every other element of $X$ to $\bot_{\mathcal{L}}$ is denoted by $\langle x \mapsto l \rangle$.

The effect of $\Omega_S$ varies according to the kind of prefixes in $S$; input prefixes only affect the $R$ component of the solution while output prefixes only affect the $K$ component. We thus define a function that operates on prefixes that handles these behaviours. The function $\Omega_\pi : \mathcal{L}_\emptyset \to \mathcal{L}_\emptyset$ (defined for $\pi$ a prefix in $\mathcal{X}$) takes a pair $(R, K) \in \mathcal{L}_\emptyset$ to the resulting pair after the effect of $\pi$ has been considered. The behaviour of the function on the two types of prefixes are given by the following auxiliary functions:

$$\begin{aligned} \Omega^{\mathcal{I}}_{(x,y)}(R, K) &= R \sqcup \langle y \mapsto K(R(x)) \rangle \\ \Omega^{\mathcal{O}}_{\langle x,y \rangle}(R, K) &= \bigsqcup_{z \in R(x)} (K \sqcup \langle z \mapsto R(y) \rangle) \end{aligned}$$

For every input prefix $(x, y) \in S$, $\Omega^{\mathcal{I}}_{(x,y)}$ adds names potentially transmitted over every name that $x$ (the channel) could assume (given by $K(R(x))$) to the names that $y$ (the datum) could itself assume. This is done by joining the given $R$ with the singleton map $\langle y \mapsto K(R(x)) \rangle$. Note that if $x$ is not itself bound by input, then

$R(x) = \{x\}$ by definition and only the set $K(x)$ is added to the image of $y$ in $R$ (i.e., only the names directly transmitted over $x$ are added). Note that this function takes a pair $(R, K)$ and only returns the resulting $R$ component (as input prefixes do not affect the $K$ component).

For any given *output* prefix $\langle x, y\rangle$, $\Omega^{\mathcal{O}}_{\langle x,y\rangle}$ augments $K$ at every name $z$ that $x$ (the channel) could assume with the set of names that $y$ (the datum) could assume, and the results are combined by the usual join operations. Note, for instance, that if neither $x$ nor $y$ are bound by input (i.e., in $\beta(S)$), then this equation reduces to adding $R(y) = \{y\}$ to the set $K(x)$. Similar to the input function, this function takes a pair $(R, K)$ and only returns the resulting $K$ component (since output prefixes do not affect the $R$ function).

Next, the functions are packaged into a function that operates on a prefix and takes a pair $\lambda = (R, K)$ to the resulting pair:

$$\Omega_\pi(\lambda) = \begin{cases} (\Omega^{\mathcal{I}}_\pi(\lambda), K) & \text{if } \pi \in \mathcal{I} \\ (R, \Omega^{\mathcal{O}}_\pi(\lambda)) & \text{if } \pi \in \mathcal{O} \end{cases}$$

Lastly, this function is extended to operate over a full set of prefixes (the representation of the desired function) to obtain an analysis step function that updates the current solution with the information gleaned from the full set. Given $S \subseteq \mathcal{X}$ a set of prefixes, the function $\Omega_S : \mathcal{L}_\emptyset \to \mathcal{L}_\emptyset$ is defined as:

$$\Omega_S(\lambda) = \bigsqcup_{\pi \in S} \Omega_\pi(\lambda)$$

For $\lambda = (R, K)$ a pair in $\mathcal{L}_\emptyset$. Observe that the operation of the prefix functions $\Omega^{\mathcal{I}}_\pi$ and $\Omega^{\mathcal{O}}_\pi$ can only *increase* the size of the image-sets of the functions $R$ and $K$ for any name, i.e., they are monotone functions over their respective lattices. It therefore follows that $\Omega_S : \mathcal{L}_\emptyset \to \mathcal{L}_\emptyset$ is a monotone function over $\mathcal{L}_\emptyset$, i.e., that $\lambda \sqsubseteq \Omega_S(\lambda)$ for any input pair $\lambda \in \mathcal{L}_\emptyset$. The Knaster-Tarski fixed point theorem therefore asserts that

this function has a least fixed point in $\mathcal{L}_\emptyset$, i.e., a least pair $\lambda$ such that $\Omega_S(\lambda) = \lambda$, and furthermore that this point can be computed by iterating $\Omega_S$ from bottom some finite number $k$ times. Thus, given a $\pi$-calculus process $P$, and recalling the representation of $P$ as the set of its prefixes $[\![P]\!]_\emptyset$ (abbreviated in the equation below as $[\![P]\!]$ for clarity), the solution $\lambda_{sol} \in \mathcal{L}_\emptyset$ to the flow and context insensitive dataflow analysis can be expressed as the least fixed point of $\Omega_{[\![P]\!]}$ in $\mathcal{L}_\emptyset$, and is computed by the following function:

$$\lambda_{sol} = (R, K) = \text{lfp}(\Omega_{[\![P]\!]}) = \Omega^{(k)}_{[\![P]\!]}(\bot_\emptyset) \tag{4.1}$$

for some sufficiently large finite integer $k$, and $h^{(k)}(x)$ denoting the application of the function $h$ to input $x$ some integer $k$ times. The number of iterations is finite, and in fact quadratic in the size of the process $P$, because there are at most $N$ sets in the solution space for each function $R$ and $K$ (with $N$ the number of names in $P$), and a maximum of $N$ names in each set.

### 4.1.3 Correctness

Correctness (with respect to a process $P$) in this setting implies that any reduction that $P$ could perform by communication is faithfully represented by the final solution pair. More formally, if $P \xrightarrow{\tau}{}^* P'$ and $P' \xrightarrow{\mu} Q'$ for some label $\mu$, then the final solution pair $(R, K)$ computed for $P$ includes the consequent of the communication $\mu$. Bodei *et al.* [BDNN01b] prove this as a subject reduction theorem for $P$ (see theorem A.1.1 in the appendix). Correctness of the approach presented above is proven here by proving that the solution $(R, K)$ computed by the abstract interpretation is a valid solution to the constraints generated by Bodei *et al.*'s analysis:

**Theorem 4.1.3.** *Given a $\pi$-calculus process $P$, let $(\rho, \kappa)$ be the least solution to the constraints generated for $P$ by the judgement in table A.1, and let $(R, K) = \text{lfp}(\Omega_{[\![P]\!]})$,*

*then*

$$(\rho, \kappa) \sqsubseteq (R, K)$$

*with* $\sqsubseteq$ *the partial ordering on solutions described in section 4.1.2.*

*Proof.* Observe first that the two solutions are elements of the same lattice $\mathcal{L}_R^{\chi(P)} \times \mathcal{L}_K$ and thus can be compared by $\sqsubseteq$. It is now possible to reason by the behaviour of $\Omega_{[\![P]\!]}$ on the two kinds of prefixes.

*Case.* **(Input)** For any input prefix $x(y)$ in $P$, the following equation holds:

$$R = R \sqcup \langle y \mapsto K(R(x)) \rangle$$

since $R$ is the least fixed point of $\Omega^{\mathcal{I}}_{(x,y)}$. By instantiating the $\sqcup$ operation to the union of the image of $y$:

$$R(y) = R(y) \cup K(R(x))$$

and therefore that $K(R(x)) \subseteq R(y)$. Expanding the application of $K$ to $R(x)$ yields

$$\bigcup_{u \in R(x)} K(u) \subseteq R(y)$$

Since the union of all such subsets $K(u)$ satisfies the inequality, each of the subsets must also satisfy the same inequality, i.e.,

$$\forall u \in R(x) : K(u) \subseteq R(y)$$

Which is the exact set of constraints satisfied by the solution pair $(\rho, \kappa)$ for $x(y)$.

*Case.* **(Output)** Similarly for any output prefix $\overline{x}\langle y \rangle$ in $P$, the following equation holds:

$$K = \bigsqcup_{z \in R(x)} K \sqcup \langle z \mapsto R(y) \rangle$$

since $K$ is the least fixed point of $\Omega^{\mathcal{O}}_{\overline{x}\langle y \rangle}$. Instantiating the $\sqcup$ operation as in the above case gives:

$$\forall z \in R(x) : K(z) = K(z) \cup R(y)$$

and therefore that $R(y) \subseteq K(z)$ for every $z \in R(x)$ Which is the exact set of constraints satisfied by $(\rho, \kappa)$ for $\overline{x}\langle y \rangle$.

Thus the solution pair $(R, K)$ satisfies the same constraints satisfied by $(\rho, \kappa)$ yielding the result. □

Note that equality does not hold in the above theorem because Bodei *et al.*'s analysis does not generate such constraints for all prefixes. Specifically, because of the rule for the match prefix in table A.1, constraints are not generated for prefixes that follow matches prefixes $[x = y]$ that don't satisfy the equation $(R(x) \cap R(y)) \neq \emptyset$. This was not included in the analysis here because in fact, this implies a context dependency in the result: a test $[x = y]$ may not pass by the reduction of $P$ as a independent process, but it may very well pass if $P$ were to run in an arbitrary context! In fact, the subject reduction theorem (theorem A.1.1) *only holds for closed processes* $P$. This drawback is further discussed and improved in the following chapter.

## 4.2 Implementation

The abstract interpretation presented above was inspired by algorithms used in optimizing compilers in order to statically predict the behaviour of references during execution of sequential programs. This section presents a demonstrated relation between "traditional" compiler-driven points-to analysis and the above analysis of the channel passing behaviour of $\pi$-calculus processes. This yields a simple polynomial time algorithm for computing the abstract execution function $\Omega_S$ discussed above.

### 4.2.1 Intuition: Andersen's Points-to Analysis

The implementation of the dataflow analysis of the $\pi$-calculus is based on Andersen's technique for doing a flow and context insensitive points-to analysis on the `C`

programming language [And99]. 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: if (cond)
3:    p = q;
4: else
5:    q = r;
6: r = &n;
```

The first step of 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, 5, and 6 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 5. 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 4.1.

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).

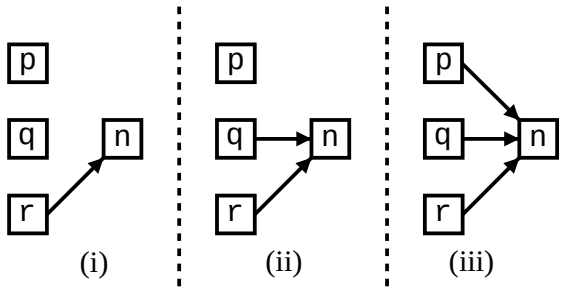



Figure 4.1: The three iterations of Andersen's analysis on our program fragment

### 4.2.2 A Simple Example

Andersen's points-to analysis is analogous to dataflow analysis in the $\pi$-calculus in that the algorithm implicitly represents a `C` program as the set of pointer assignment statements it contains (much as the dataflow analysis represents a process $P$ as the set of prefixes it contains), and then updates the solution function `pts` based on a flow equation. This iterative technique can be applied to $\pi$-calculus processes in an effort to algorithmically compute the function $\Omega_S$. This is done by coming up with an iterative algorithm for each of the auxiliary functions $\Omega_S^{\mathcal{I}}$ and $\Omega_S^{\mathcal{O}}$. Luckily, this is quite easy to do. Observe that the solution pair $(R, K) \in \mathcal{L}_\emptyset$ can only take names in the process $P$ to a non-bottom value, thus the solutions are implemented by two tables `R` and `K` (distinguished from the abstract functions $R$ and $K$) of $O(N)$ entries, with $N > |\mathrm{n}(P)|$ the number of symbols in $P$, which can each map to a set of at most $O(N)$ elements. Thus only $O(N^2)$ space is required to store the solution. The `K` table is initialized to map each element to the empty set, while the `R` table is initialized to map every element to a set containing only itself (representing the $\bot_K$ and $\bot_R$ functions respectively).

Next, recall that the $\sqcup$ operation in the $\Omega_S^{\mathcal{I}}$ function is a simple set union: for any set $S$, given any function $F$ in $[S \Rightarrow 2^S]$, the supremum of $F$ with a singleton map $\langle y \mapsto Y \rangle$ (for $y \in S$ and $Y \in 2^S$) is the same function as $F$ with every element of $Y$

added to the image $F(y)$, i.e.,:

$$F \sqcup \langle y \mapsto Y \rangle = F\{y \mapsto (Y \cup F(y))\}$$

In the case where $F$ is implemented by a table (as in `R` and `K`), this can be computed by simply merging the set $F(y)$ (obtained with a table lookup) with the set $Y$, a quadratic operation in the worst case. Furthermore, because set unions are commutative and associative, they can be done in any order, and thus the functions $\Omega_S^{\mathcal{I}}$ and $\Omega_S^{\mathcal{O}}$ can be implemented by iterating through their respective prefix sets in any order, and adding the appropriate elements for each prefix.

This section offers an example of the procedure outlined above in action, consider the following $\pi$-calculus process:

**Example 4.2.1.**

$$P \triangleq a(x).(\nu b)(\nu c)((\overline{b}\langle a\rangle.\overline{x}\langle x\rangle.b(y).\overline{y}\langle c\rangle + !\overline{b}\langle d\rangle.\overline{a}\langle c\rangle) \parallel b(z).\overline{b}\langle z\rangle) \parallel d(w)$$

This is identical to example 3.1 in [BDNN01b] for comparison, except that the extensions have been removed and the process has been $\alpha$-converted in order to eliminate ambiguities in naming. Note that $\beta(P) = \{x, y, z, w\}$ and that $\mathrm{n}(P) \setminus \beta(P) = \chi(P) = \{a, b, c, d\}$. Note that the full filtered input functionality is not used for clarity. Analogous to Andersen's analysis where all statements are stripped of the program except for pointer assignments, only input and output prefixes are considered here. Specifically, the representation of $P$ is computed as $[\![P]\!]_\emptyset = \mathcal{X}_P = \mathcal{I}_P \cup \mathcal{O}_P$ where $\mathcal{I}_P = \{a(x), b(y), b(z), d(w)\}$ and $\mathcal{O}_P = \{\overline{b}\langle a\rangle, \overline{x}\langle x\rangle, \overline{y}\langle c\rangle, \overline{b}\langle d\rangle, \overline{a}\langle c\rangle, \overline{b}\langle z\rangle\}$. The tables `K` and `R` are next initialized appropriately, and these are updated by iterating through the list of prefixes. The step-by-step evolutions of the analysis solution is depicted in table 4.2.2.

In the first iteration, the procedure (while computing $\Omega^{\mathcal{I}}$) encounters the input prefix $a(x)$. $R(a) = \{a\}$ initially, thus $R(x)$ need only be updated with the set of

| | | | $a(x)$ | $b(y)$ | $b(z)$ | $d(w)$ | $\overline{b}\langle a\rangle$ | $\overline{x}\langle x\rangle$ | $\overline{y}\langle c\rangle$ | $\overline{b}\langle d\rangle$ | $\overline{a}\langle c\rangle$ | $\overline{b}\langle z\rangle$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **I** | $R$ | $x$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ |
| | | $y$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ |
| | | $z$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ |
| | | $w$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ |
| | $K$ | $a$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{c\}$ | $\{c\}$ |
| | | $b$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{a\}$ | $\{a\}$ | $\{a\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ |
| | | $c$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ |
| | | $d$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ |
| **II** | $R$ | $x$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ |
| | | $y$ | $\{\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ |
| | | $z$ | $\{\}$ | $\{\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ |
| | | $w$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ |
| | $K$ | $a$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ |
| | | $b$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ |
| | | $c$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ |
| | | $d$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ |
| **III** | $R$ | $x$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ |
| | | $y$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ |
| | | $z$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ |
| | | $w$ | $\{\}$ | $\{\}$ | $\{\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ |
| | $K$ | $a$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ |
| | | $b$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ | $\{a,d\}$ |
| | | $c$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ |
| | | $d$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ |

Table 4.1: The evolution of the analysis solution $(R, K)$ as each prefix is encountered over the three iterations of the algorithm

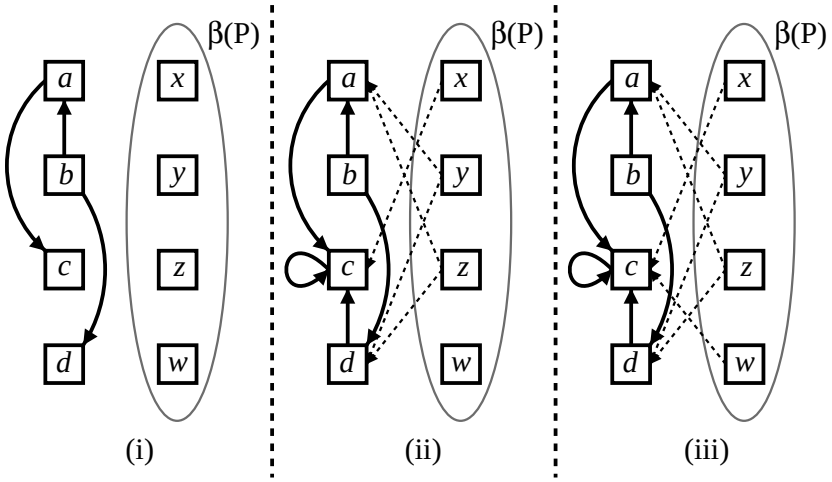



Figure 4.2: The three iterations of the computation of $\Omega^{(k)}_{[\![P]\!]}(\bot_\emptyset)$

possible values that $a$ could transmit, which by definition is currently $K(a) = \emptyset$. Thus, no new information is gained from the prefix $a(x)$, and similarly for the other input prefixes. The computation of $\Omega^{\mathcal{O}}$ begins with the prefix $\overline{b}\langle a\rangle$, which yields that $K(b) = \{a\}$ since neither $a$ nor $b$ are bound *by input*. The next prefix ($\overline{x}\langle x\rangle$) does not yield any new data because $x \in \beta(P)$, and $R(x) = \emptyset$. Similarly, the prefixes $\overline{y}\langle c\rangle$ and $\overline{b}\langle z\rangle$ do not change the solution since there is no information about the input bound names $y$ and $z$. The prefixes $\overline{b}\langle d\rangle$ and $\overline{a}\langle c\rangle$, however, do tell us that $K(b)$ must include $\{d\}$ and that $K(a)$ must include $\{c\}$ respectively. Hence, at the end of the first iteration, we have that $K(b) = \{a, d\}$ and $K(a) = \{c\}$. The solution pair representing $\Omega_{[\![P]\!]}(\bot_\emptyset)$ is presented as a graph (similar to the graph generated for the points-to example) in figure 4.2(i). Note that this figure presents the values of the function $K$ as solid arrows, and the values of $R$ are shown as dotted arrows (the self loops representing $R(a) = \{a\}$ for the names in $\chi(P)$ are omitted for clarity).

The second iteration begins again by going through the list of input prefixes, but this time, there is more information available than $\bot_\emptyset$. The prefix $a(x)$ at this point tells us that anything transmitted over $a$ must be part of the names received by $x$, thus

$R(x) = \{c\}$ after seeing this prefix because $K(a) = \{c\}$. Similarly, the prefixes $b(y)$ and $b(z)$ assign everything in the $K$ set of $b$ to the $R$ set of $z$ yielding $R(y) = \{a, d\}$ and $R(z) = \{a, d\}$. The last input prefix $d(w)$ yields no new information because there is 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 $\overline{x}\langle x\rangle$, $\overline{y}\langle c\rangle$, and $\overline{b}\langle z\rangle$. The first indicates that anything that $x$ may have received (which is currently $R(x) = \{c\}$) could be transmitted over itself, thus $K(c) = \{c\}$. The second says that $c$ could be transmitted over anything that $y$ may have received ($R(y) = \{a, d\}$), hence only $c$ is added to $K(d)$ since $c \in K(a)$ already. The third prefix ($\overline{b}\langle z\rangle$) says that anything that $z$ may have received could be transmitted over $b$, so all elements of $R(z) = \{a, d\}$ must be added to $K(b)$. But $a$ and $d$ are already in this set so nothing needs to be done. The solution $\Omega^{(2)}_{[\![P]\!]}(\bot_\emptyset)$ after the second iteration is presented graphically in figure 4.2(ii).

Finally, the third iteration of the function only adds the element $c$ to $R(w)$, and the fourth iteration makes no change indicating that the fixed point has been reached. The resulting graph representing the solution after the third iteration of the analysis on $P$ is shown in figure 4.2(iii). It is easy to check that $\Omega^{(4)}_{[\![P]\!]}(\bot_\emptyset) = \Omega^{(3)}_{[\![P]\!]}(\bot_\emptyset)$, and thus this is the final solution of the analysis.

In terms of efficiency, the trivial iterative algorithm takes $O(N^6)$ time to compute (quadratic iterations and a worse case time of $O(N^4)$ in order to do the unions and intersections required for each iteration), but this can be improved by expressing the analysis as a set of Horn Clauses (see [NS01]) to compute the function $\Omega_{[\![P]\!]}$ in cubic time. A similar procedure could be used to compute all subsequent flow analyses discussed in this thesis, as these in general involve restricting the prefix set that the function operates on based on various conditions. As long as the conditions are checkable in polynomial time, then the analysis stays polynomial.

Again, notice the context-dependent nature of this result: while it has been computed that the name $w$ can only have $R(w) = \{c\}$, this is not necessarily true in any context. If $P$ were to interact with a process $C = \overline{d}\langle e \rangle$ for some name $e$ (i.e., if $P$ were plugged into the context $C \parallel [\cdot]$, then $R(w)$ would also need to include $e$ in order to be a correct result. It is thus evident that the analysis results generated in this chapter are only correct if the process $P$ analyzed runs in isolation. This limits the utility of the analysis for applications such as security (because protocols often run in hostile environments), thus the next chapter is devoted to generating results that consider the role of the potential environment that the process could run in.

# Chapter 5
# Tracking Environment Knowledge

The analysis of the previous chapter does not account for the environment in which the process under scrutiny could run. This implies that the analysis results are only correct for closed processes (see the simple example at the end of the previous chapter). This severely limits the usefulness of such control flow analyses for many applications, notably for security, as the properties proved don't apply in every context. For instance, Bodei *et al.* [BDNN01b] discuss the security property of a process $P$ having "no leaks", which essentially formalizes the notion that a name bound to $P$ stays confined to $P$. However, the analysis that is described there does not assume that all matches in $\mathcal{M}_P$ pass (as the one described in the previous chapter does). Thus the contribution of a prefix following a match that doesn't succeed is not included in the final solution. Unfortunately, this is not necessarily true, consider the following process:

$$P \triangleq a(x).b(y).[x = y].(\nu c)(\nu d)(\overline{d}\langle c\rangle \| d(z).\overline{a}\langle z\rangle)$$

The analysis presented in [BDNN01b] (and in fact any other current analyses of the $\pi$-calculus) would infer that $[x = y]$ can not possibly pass, and thus the computation of the subsequent subprocess (which shall hereafter be shortened to $P'$) could not occur.

It would then be inferred that this process has no leaks (i.e., none of its bound names are extruded). However, consider the interaction of $P$ with the following process:

$$C \triangleq \overline{a}\langle e\rangle.\overline{b}\langle e\rangle.a(w).C'$$

In other words, consider the reduction sequence of $C\|P$ (reduced redices are underlined at each step):

**Example 5.0.2.**

$$\begin{array}{rcrcl}
C\|P & \triangleq & \overbrace{\underline{\overline{a}\langle e\rangle}.\overline{b}\langle e\rangle.a(w).C'}^{C} & \| & \overbrace{\underline{a(x)}.b(y).[x=y].P'}^{P} \\
& \xrightarrow{\tau} & \underline{\overline{b}\langle e\rangle}.a(w).C' & \| & \underline{b(y)}.[e=y].P'\{e\mapsto x\} \\
& \xrightarrow{\tau} & a(w).C' & \| & \underline{[e=e]}.P'\{e\mapsto x\}\{e\mapsto y\} \\
& \xrightarrow{\tau} & a(w).C' & \| & (\nu c)(\nu d)(\underline{\overline{d}\langle c\rangle}\|\underline{d(z)}.\overline{a}\langle z\rangle) \\
& \xrightarrow{\tau} & \underline{a(w)}.C' & \| & (\nu c)(\nu d)(\mathbf{0}\|\underline{\overline{a}\langle c\rangle}) \\
& \xrightarrow{\tau} & (\nu c)(C'\{w\mapsto c\} & \| & (\nu d)(\mathbf{0}\|\mathbf{0}))
\end{array}$$

Observe that in this context, the private name $c$ has indeed been leaked out of $P$ to the environment $C$. This example not only illustrates the fact that a treatment of the environment is vital to proper security analysis, but also that such a treatment isn't completely trivial. It is not sufficient to assume that any name not in $P$ could be received, as names bound to $P$ can exit the scope of $P$, and such leaks must be tracked. This chapter endeavours to add this feature to the dataflow analysis.

## 5.1 Expansion of the Solution Space

The analysis described in the previous chapter produces a solution in the lattice $\mathcal{L}_\emptyset^P = \mathcal{L}_R^{\chi(P)} \times \mathcal{L}_K$ for a given process $P$. This solution space is extended with a third component $E \subseteq \mathcal{N}$ that keeps track of the set of names that the environment knows about, and is thus called the *environment knowledge*. Bodei *et al.* [BDPZ03] provided

a partial treatment of this component, which is expanded (and proved correct) in this chapter. Observe first that the set of potential solutions for $E$ is just the power set of $\mathcal{N}$, i.e., $2^{\mathcal{N}}$, and thus forms a complete lattice with $\emptyset$ as bottom, and $\mathcal{N}$ as top. An initialization property similar to proposition 4.1.2 for the $R$ component applies here as well:

**Proposition 5.1.1.** *Given a countable set $S$, let $\mathcal{L}$ denote the usual lattice of subsets of $S$, and consider any subset $I \subseteq S$, the set $\mathcal{L}_I = \{T \in \mathcal{L} \mid I \sqsubseteq T\}$ of subsets of $S$ that include $I$ forms a complete sub-lattice of $\mathcal{L}$ with $I$ as bottom, and $\mathcal{N}$ as top.*

The proof is trivial, and closely follows the proof of proposition 4.1.2, and is thus omitted. Just as for the $R$ component, this is used to "initialize" $E$ for a given process $P$.

This initialization is done by defining the special name $\xi_P$ to denote the set of names that do not appear in $P$, i.e., $\xi_P = \mathcal{N} \setminus \mathrm{n}(P)$. Inclusion or non-inclusion in this infinite set can be tested easily by testing for non-inclusion or inclusion in $\mathrm{n}(P)$ respectively. This "set", together with the set of free names of $P$, will act as the initialization set for the environment knowledge component $E$ of the solution: it describes the set of names that the environment of a process $P$ may know about before it begins its reduction sequence.

The new solution lattice is denoted by $\mathcal{L}_{\mathcal{E}}^P$ and defined as the triple $\mathcal{L}_{\mathcal{E}}^P = \mathcal{L}_R^{\chi(P)} \times \mathcal{L}_K \times \mathcal{L}_E^{\xi_P \cup \mathrm{fn}(P)}$, where $\mathcal{L}_E^{\xi_P \cup \mathrm{fn}(P)}$ denotes the sub-lattice of $2^{\mathcal{N}}$ comprising the elements that include the set $(\xi_P \cup \mathrm{fn}(P))$, as constructed by proposition 5.1.1, and therefore having bottom element $(\xi_P \cup \mathrm{fn}(P))$ denoted $\bot_E$. Once again, the process-dependent superscripts $P$, $\chi(P)$, and $\xi_P \cup \mathrm{fn}(P)$ are omitted when the process in question is clear from the context.

### 5.1.1 Abstract Execution

Using the same abstract representation $[\![P]\!]_\emptyset$ of a process $P$, it is possible to modify the abstract execution function $\Omega_S$ in order to obtain a fully context independent analysis. Thus the input component will again take a solution triple $(R, K, E)$ to the next $R$ component. Furthermore, given an input prefix $x(y)$, all the names known to the environment must be added to $R(y)$ if any of the names that $x$ could assume (i.e., $R(x)$) are in the environment. Formally, letting $\lambda = (R, K, E)$, the modified prefix function $\Phi^{\mathcal{I}}_{(x,y)}$ is now defined as:

$$\Phi^{\mathcal{I}}_{(x,y)}(\lambda) = \begin{cases} \Omega^{\mathcal{I}}_{(x,y)}(R, K) \sqcup \langle y \mapsto E \rangle & \text{if } (R(x) \cap E) \neq \emptyset \\ \Omega^{\mathcal{I}}_{(x,y)}(R, K) & \text{otherwise} \end{cases}$$

In words, the function is identical to the corresponding function in the previous analysis, unless the environment may have knowledge of the communication channel $x$ (determined by the condition $(R(x) \cap E) \neq \emptyset$), in which case it is conservatively assumed that any name in the environment knowledge could potentially be received, and thus must be added to $R(y)$. The Greek letter $\Phi$ is used as mnemonic for PHlow Insensitive.

In contrast, an output prefix can actually expand the knowledge of the environment by transmitting a private name to it. Specifically, given an output prefix $\overline{x}\langle y \rangle$, the environment knowledge is expanded with $R(y)$ if the environment knows about any name in $R(x)$. Thus, the modification of the output component must take a solution triple $\lambda = (R, K, E)$ to a pair $(K', E')$ representing the new transmission component $K'$ *as well as* the expanded environment knowledge $E'$. Formally, the $\Omega^{\mathcal{O}}_{\langle x,y \rangle}$ function is modified to $\Phi^{\mathcal{O}}_{\langle x,y \rangle}$ as follows:

$$\Phi^{\mathcal{O}}_{\langle x,y \rangle}(\lambda) = \begin{cases} (\Omega^{\mathcal{O}}_{\langle x,y \rangle}(R, K), E \cup R(y)) & \text{if } (R(x) \cap E) \neq \emptyset \\ (\Omega^{\mathcal{O}}_{\langle x,y \rangle}(R, K), E) & \text{otherwise} \end{cases}$$

This is identical to the previous definition, except that the expansion of the environment is performed as described above. Since the environment is always expanded, and the function is otherwise identical to the previous definition, this function is also monotone.

This function can again be packaged into a function $\Phi_\pi$ operating over any prefixes and taking any triple $\lambda = (R, K, E) \in \mathcal{L}_\mathcal{E}$ as input:

$$\Phi_\pi(\lambda) = \begin{cases} (\Phi^\mathcal{I}_\pi(\lambda), K, E) & \text{if } \pi \in \mathcal{I} \\ (R, \Phi^\mathcal{O}_\pi(\lambda)) & \text{if } \pi \in \mathcal{O} \end{cases}$$

Observe that the equation above actually generates a pair of the form $(R, (K, E))$ for output prefixes, but these are taken to be triples $(R, K, E)$ in a clarifying abuse of notation. Lastly, this can again be trivially extended to a function $\Phi_S$ that operates with $S$ a set of prefixes just as for $\Omega_S$:

$$\Phi_S(\lambda) = \bigsqcup_{\pi \in S} \Phi_\pi(\lambda)$$

The full dataflow analysis solution $\lambda_{sol} = (R, K, E)$ can thus be expressed as the least fixed point of $\Phi_S$ in $\mathcal{L}_\mathcal{E}$ computed by the following iteration:

$$\lambda_{sol} = (R, K, E) = \text{lfp}(\Phi_{[\![P]\!]}) = \Phi^{(k)}_{[\![P]\!]}(\bot_\mathcal{E}) \tag{5.1}$$

where the bottom element $\bot_\mathcal{E}$ denotes the triple $(\bot_R, \bot_K, \bot_E)$ corresponding to the process being analyzed. It is obvious that given a set of prefixes $S$, the above function is of type $\Phi_S : \mathcal{L}_\mathcal{E} \to \mathcal{L}_\mathcal{E}$, and its monotonicity implies that its least fixed point exists and can be computed by the iteration above with $k \leq |P|^2$. Furthermore, since the condition $(R(x) \cap K(x) \neq \emptyset)$ is polynomially checkable for a finite process, the entire analysis remains computable in polynomial time.

**Example 5.1.1.** Revisiting example 4.2.1:

$$P \triangleq a(x).(\nu b)(\nu c)((\overline{b}\langle a\rangle.\overline{x}\langle x\rangle.b(y).\overline{y}\langle c\rangle + !\overline{b}\langle d\rangle.\overline{a}\langle c\rangle) \parallel b(z).\overline{b}\langle z\rangle) \parallel d(w)$$

Recall that

$$\begin{aligned} \mathrm{fn}(P) &= \{a, d\} \\ \beta(P) &= \{x, y, z, w\} \\ \chi(P) &= \{a, b, c, d\} \end{aligned}$$

Thus $\perp_E = \mathrm{fn}(P) \cup \xi_P = \{a, d, \xi_P\}$, by the definition of $\perp_E$. Note that the free channels $a$ and $d$ are initially known to the environment. The new function $\Phi_{[\![P]\!]}$ performs the same computations as $\Omega_{[\![P]\!]}$, except for the following instances:

- In the first iteration of $\Phi_{[\![P]\!]}$, the output component must account for the fact that the prefix $\overline{a}\langle c\rangle$ transmits the bound name $c$ on the free channel $a$. This is detected by the fact that $(R(a) \cap E) = \{a\} \neq \emptyset$, thus the names $R(c) = \{c\}$ must be added to the environment knowledge by the definition of $\Phi^{\mathcal{O}}$.

- The second iteration now must compute in the context that the bound name $c$ may be a part of the environment knowledge. Therefore, the input prefixes $a(x)$ and $d(w)$ now imply that $R(x)$ and $R(w)$ must include $E = \{a, d, c, \xi_P\}$ *in addition to* all of the names in $R(a)$ and $R(d)$ respectively.

The final solution of the analysis is as follows (with the self-loops for the $R(u) = \{u\}$ for $u \in \chi(P)$ omitted for clarity):

$$\begin{aligned} R(x) &= \{a, d, c, \xi_P\} & K(a) &= \{a, d, c, \xi_P\} \\ R(y) &= \{a, d\} & K(b) &= \{a, d\} \\ R(z) &= \{a, d\} & K(c) &= \{a, d, c, \xi_P\} \\ R(w) &= \{a, d, c, \xi_P\} & K(d) &= \{a, d, c, \xi_P\} \end{aligned}$$

And the final environment knowledge $E = \{a, d, c, \xi_P\}$. Contrast this with the final solution of $\Omega_{[\![P]\!]}$ in figure 4.2(iii), and it is obvious that the only difference is that every set that contained $c$ (the extruded bound name) in the previous solution now also contains the initial environment knowledge $\perp_E$.

### 5.1.2 Correctness

Due to the fact that the potential knowledge of the environment is now being tracked by the analysis, the correctness result that can be proven will apply to the analyzed process running concurrently with any environment. Before this is done however, it is useful to prove a lemma relating the current analysis to the previous context-dependent one:

**Lemma 5.1.2.** *Let $P$ be a closed $\pi$-calculus process, and let $(R, K) = \mathrm{lfp}(\Omega_{[\![P]\!]})$ and $(R_e, K_e, E_e) = \mathrm{lfp}(\Phi_{[\![P]\!]})$, then*

$$(R, K) = (R_e, K_e)$$

*where equality is taken in the $\mathcal{L}_\emptyset$ lattice.*

*Proof.* Observe that if $P$ is closed, the environment $E_e$ is initialized to $\xi_P$, i.e., the environment only knows about the names not appearing in $P$ as expected. This makes it impossible for the condition $(R_e(x) \cap E_e) \neq \emptyset$ to pass in any iteration of $\Phi_{[\![P]\!]}$. Thus, by the definition of $\Phi$, $E_e$ is never modified and $\Phi$ behaves as $\Omega$ yielding the result. □

The full proof of correctness that applies over any process cannot rely on the constraint based analysis of Bodei *et al.* to establish its result because the latter is context dependent. In order to use that technique, a new flow logic would need to be defined and proved correct in order to prove the validity of the solution. This technique is avoided in this thesis in favor of a more direct approach whereby the result of the analysis is compared to the actions that could occur in all possible traces of a process. The justifications for not adapting Bodei *et al.*'s technique here are twofold:

1. A flow-logic approach abstracts the constructive algorithm and entwines the algorithm description with the proof of correctness. This thesis is intended to be read so that proofs of correctness can be skipped if desired, and the present approach allows the dissociation of algorithm from proof which allows this to occur.

2. A new approach to a correctness proof may provide new insight about the inner workings of the dataflow analysis.

In order to prove correctness, first observe that there are only three possible ways that a process $P$ can reduce:

1. A silent prefix $\tau$ can be removed (rule `(R Tau)` in table 3.3).

2. A match prefix $[x = x]$ can be removed (rule `(R Match)` in table 3.3).

3. A communication between an input and output prefix in parallel composition (rules `(R Com)` and `(R Close)` from table 3.3).

Furthermore, observe that no name can be substituted for more than once: if it is bound by an input prefix at some program point, the same name cannot appear bound in some later subprocess because of assumption 1.

To adequately compare a dataflow analysis solution to an actual execution trace of a process, a mechanism is required to retrieve the original names (i.e., those given as the input to the dataflow analysis algorithm) appearing in each prefix occurence. This is because the names in any given prefix may change through substitution as the process executes.

**Definition 5.1.1.** Let $\pi \in \mathcal{X}_P$ denote the occurence of a prefix in a process $P$ given as an input to the dataflow analysis algorithm, and let $\pi' \in \mathcal{X}_P$ denote the same prefix occurence some number of steps into the execution of $P$. Suppose that name

substitutions during executions may have caused $\text{cha}[\pi'] \neq \text{cha}[\pi]$, or $\text{dat}[\pi'] \neq \text{dat}[\pi]$, or both. Then define the function $\sigma : \mathcal{X}_P \to \mathcal{X}_P$ by

$$\begin{aligned} \text{cha}[\sigma(\pi')] &= \text{cha}[\pi] \\ \text{dat}[\sigma(\pi')] &= \text{dat}[\pi] \end{aligned}$$

i.e., the function $\sigma$ returns the prefix as it appeared in $P$ prior to execution.

In order to precisely define the dataflow that actually occurs during the reduction of a process, a *communication action* $\alpha$ is defined by an input/output pairing of prefixes $\alpha = (\pi_{\mathtt{i}}, \pi_{\mathtt{o}})$. Such an action completely describes a single-step communication in the execution sequence of a process. For example, given the process $P \triangleq a(x).\overline{x}\langle c\rangle \| \overline{a}\langle b\rangle$ the only possible action is $\alpha = (a(x), \overline{a}\langle b\rangle)$, or using the pairs notation for prefixes:

$$\alpha = ((a, x), \langle a, b\rangle)$$

This action represents the name $b$ being sent over the channel $a$, and saved into $x$. As an example of the usage of the $\sigma$ mapping, consider $P$ after the action $\alpha$ is taken, i.e., the process $\overline{b}\langle c\rangle$. If $P$ were given as the input to the flow analysis, then by definition:

$$\sigma(\overline{b}\langle c\rangle) = \overline{x}\langle c\rangle$$

because the occurence of the prefix $\overline{b}\langle c\rangle$ was originally $\overline{x}\langle c\rangle$ in $P$ prior to the action $\alpha$.

This definition of actions excludes $\tau$ and match actions as described above because these do not affect dataflow in any way. It is now possible to define a trace of a process as a sequence of actions:

**Definition 5.1.2.** Given a process $P$, a *trace* of $P$ is a (possibly infinite) sequence of reductions $P \xrightarrow{\tau} \ldots$ that $P$ may take. Each reduction step that involves a communication can be labelled by a communication action as defined above. A *dataflow trace*

of a process $P$ can be formalized by a (potentially infinite) sequence of such actions $\alpha_1, \alpha_2, \ldots$.

The original process $P$ is assumed to be $\alpha$-converted to avoid repeated bound names as above, but it is then assumed that no further $\alpha$-conversions occur during the execution of $P$. This condition is imposed to preserve the names that occur inside replicated sub-processes; i.e., if the reduction applies the structural congruence rule `(S Rep)` to the process $!(\nu a)P$ to obtain $(\nu a)P \| !(\nu a)P$, it is assumed that the name $a$ is preserved in the unfolded process instance. This name preservation is an approximation of the actual behaviour of $P$ that provides the following features useful for the purpose of proving correctness:

- The trace can be compared with the result of the dataflow analysis, which does not account for the distinction between the infinite bound names in a replicated process.
- The set of possible actions becomes finite for a given process $P$, implying that infinite traces are only infinite because they repeat actions.

The second point here requires some elaboration. For any process $P$, the syntactic set of names $\mathrm{n}(P)$ is finite because the process term is finite. As noted above, each bound name in a replicated process represents an infinite set of semantically distinct names. The approximation above identifies all of these names, thus the set of names of a process is kept distinct, the number of substitutions between them is also finite, and hence the set of actions of the process that can occur during reduction is also finite. Thus, any infinite reduction sequence in this model implies an infinite number of repeated actions.

Correctness is proved by defining a function $\Lambda^R$ is taking traces to a dataflow solution $R$. This function essentially defines an approximate (because of the name

identification above) dataflow solution that is known to be correct (because it is built from the actual traces of the process $P$). Correctness is proved by showing that the analysis solution is at least as great as this function. The function $\Lambda^R$ is first defined as operating on a single action:

$$\Lambda^R(\alpha) = \langle \mathrm{dat}[\pi_{\mathtt{i}}] \mapsto \{\mathrm{dat}[\pi_{\mathtt{o}}]\}\rangle$$

where

$$\alpha = (\pi_{\mathtt{i}}, \pi_{\mathtt{o}})$$

is an action with $\pi_{\mathtt{i}}$, $\pi_{\mathtt{o}}$ an input and output prefix respectively. Notice that this function returns a function in $\mathcal{L}_R$ mapping the datum of the input prefix to the datum of the ouput prefix, i.e., it records the the actual communication as a function in $\mathcal{L}_R$. For example, letting $\alpha = (a(x), \overline{a}\langle b\rangle)$ be the action from the above example then by definition:

$$\Lambda^R(\alpha) = \langle x \mapsto \{b\}\rangle$$

i.e., $R(x) = \{b\}$. In words, this encodes the fact that the name $b$ was transmitted and stored into the name $x$. The function $\Lambda^R$ is then compositionally extended to sets of traces by combining the results of individual actions over all traces. Formally, letting $S$ be a set of traces, $tr$ an individual trace, and $\alpha$ an action:

$$\Lambda^R(S) = \bigsqcup_{tr \in S} \bigsqcup_{\alpha \in tr} \Lambda^R(\alpha)$$

Now, observe that $\Lambda^R(\alpha)$ is only locally dependent on the action $\alpha$. Thus, repeated actions in a trace have no effect on the result of the $\Lambda$ function. Thus the effect of $\Lambda^R$ over some infinite trace $tr_\infty$ is identical to the result computed by computing $\Lambda^R$ over the *finite* trace resulting from removing all repeated actions from $tr_\infty$. Removing repeated actions results in a finite trace because the only source of repeated actions (keeping in mind that each action refers to syntactic *occurrences* of prefixes) is a

replicated process. The set of all such finite traces generated from a process $P$ is denoted $\mathbf{H}[P]$.

Note that only the $R$ component of the interaction is recorded. The context independence proof only compares this component of the generated dataflow analysis with the actual solution generated by $\Lambda^R$. This feature simplifies the analysis as well as the proof because it does not require the analysis to account for the possibility of the context transmitting over channels that $P$ does not transmit over. Formally, if $x$ were bound in $P$, but later extruded to the environment, the analysis would need to account for the fact that any name known to the environment could be transmitted over $x$ (through the presence of an output prefix $\pi$ with $\mathrm{cha}[\pi] = x$ appearing in the context). The analysis as it stands does not do this, but doing so would require an extension of the function $\Phi_S^{\mathcal{O}}$ that would add the names in the environment to the $K$ set of every name that the datum of the output prefix could assume. Intuitively, this would account for the cases where a given communication occurs in the environment.

This aspect is omitted from the results presented here in order to keep the analysis as simple as possible, to simplify the proof of correctness, and because there is a philosophical issue regarding whether such information is actually useful (see the concluding chapter 8). Consider, for instance, a process $P$ that is comprised of a single input prefix $b(x)$. The current analysis would compute that the set of names that $x$ could become (i.e., $R(x)$) could be $b \cup \xi_P$ because the name $b$ is free in $P$ and could thus receive input from the environment. However, the dataflow analysis computes that no names could be transmitted by $b$ (i.e., $K = \emptyset$). Unfortunately, this is not necessarily true if the context uses $b$ to transmit names without interacting in any way with $P$. This leads to the question of whether it is useful in any way to keep track of such information by initializing $K(x)$ to $E$ for every free name in the analyzed process, and also performing this action when a bound name is extruded to the environment. The position taken by this thesis answers that question in the

negative, the justification being that such actions do not discuss the dataflow actions of the the analyzed process.

Thus, the $K$ component is compared to the result of a function $\Lambda_C^K$, which records the transmitted name in each action in the traces into a function in $\mathcal{L}_K$, but excludes those actions whose name transmissions occurs in the process environment $C$. Formally, the operation of $\Lambda_C^K$ on an action $\alpha = (\pi_{\mathtt{i}}, \pi_{\mathtt{o}})$ is as follows:

$$\Lambda_C^K(\alpha) = \begin{cases} \langle \mathrm{cha}[\pi_{\mathtt{i}}] \mapsto \{\mathrm{dat}[\pi_{\mathtt{o}}]\} \rangle & \text{if } \pi_{\mathtt{o}} \notin \mathcal{X}_C \\ \bot_K & \text{otherwise} \end{cases}$$

This simply says that the transmission is recorded only if the output prefix is not in the context (and in fact is in the analyzed process). The results can again be combined over all actions in the traces as above.

A similar function can be computed for the $E$ component, which only records the result if the the action involves a prefix not in $C$ sending a value to a prefix in $C$ (i.e., if the action involved the transmission of a name from the analyzed process to the environment). This function is defined in the obvious way and denoted $\Lambda_C^E$.

A context is modelled by an arbitrary $\pi$-calculus process with which $P$ may run in parallel, i.e., the result of an analysis of a process $P$ is proved as correctly predicting the dataflow in $C\|P$ for any process $C$. Similar results for arbitrary contexts $C[\cdot]$ are left for future work (see chapter 8).

The correctness of a dataflow analysis can now be established by showing that the analysis solution $R$ (for some process $P$) is greater than $\widehat{R} = \Lambda^R(\mathbf{H}[C\|P])$ in the lattice $\mathcal{L}_R$ for any process $C$, and the analysis solution $K$ (for some process $P$) is greater than $\widehat{K} = \Lambda_C^K(\mathbf{H}[C\|P])$. However this comparison cannot be made directly because $\widehat{R}$ may be defined over names that appear only in the context. Thus, a restriction operator $\nabla$ is defined to remove these names from the domains of the trace solution $\widehat{R}$. Specifically, given a function $R \in \mathcal{L}_R$, the function $R\nabla P$ is defined

as follows:

$$R \nabla P = \bigsqcup_{x \in \mathrm{n}(P)} \langle x \mapsto R(x) \rangle$$

i.e., the values of the functions over the names in $P$ are preserved, and all other names are mapped to $\emptyset$.

One final notion needs to be introduced before the correctness result can be proved. Consider an action $\alpha_0 = (\pi_{\mathtt{i}}, \pi_{\mathtt{o}})$ in a trace $tr$. Each name in the pairs $\sigma(\pi_{\mathtt{i}})$ and $\sigma(\pi_{\mathtt{o}})$ that is different from the names in $\alpha_0$ implies an action that precedes $\alpha_0$ in $tr$ that caused the substitution to occur. If the substitutions in each of these preceding actions are themselves not empty, then this process can be repeated to find preceding actions for each of them. The tree generated by iterating this procedure is called the *dependency tree* of action $\alpha_0$, and is finite for any process $P$ where the trace $tr$ is taken from $\mathbf{H}[P]$. The tree is finite because the trace is finite and each child always goes to a previous action in the trace. The leaves of the tree are actions where the prefixes in the action appear in their original form (prior to execution).

The proof of correctness of the dataflow analysis reasons on the structure of the dependency tree to establish its result. The proof requires a few preliminary lemmas that establish the correct behaviour of $\Phi$ on individual actions, and sub-trees of the dependency tree of an action in $\mathbf{H}[C \| P]$. First, a few basic definitions are presented:

**Definition 5.1.3.** Given an action $\alpha$ in $\mathbf{H}[C \| P]$ for some process $P$ and some context $C$. The set of prefixes appearing in the dependency tree of $\alpha$ is denoted $\mathcal{X}(\alpha)$. The subset of this set comprising prefixes occurring in $P$ is denoted $\mathcal{X}_P(\alpha)$.

The following lemma establishes the correctness of the analysis at the leaves of the dependency trees in a trace:

**Lemma 5.1.3.** *Given an action $\alpha = (\pi_{\mathtt{i}}, \pi_{\mathtt{o}})$ in $\mathbf{H}[C \| P]$, for some process $P$ and*

*context $C$, such that*

$$\begin{aligned}\sigma(\pi_{\mathtt{i}}) &= \pi_{\mathtt{i}} \\ \sigma(\pi_{\mathtt{o}}) &= \pi_{\mathtt{o}}\end{aligned}$$

*i.e., $\alpha$ has no children in the dependency tree. Furthermore, let*

$$(\widehat{R}, \widehat{K}, \widehat{E}) = (\Lambda^R(\alpha), \Lambda^K_C(\alpha), \Lambda^E_C(\alpha))$$

*and*

$$(R, K, E) = \mathrm{lfp}(\Phi_{\mathcal{X}_P(\alpha)})$$

*then*

$$(\widehat{R}\nabla P, \widehat{K}, \widehat{E}) \sqsubseteq (R, K, E)$$

*Proof.* Since $\alpha$ is in $\mathbf{H}[C\|P]$, and has no dependencies, it is known that $\mathrm{cha}[\pi_{\mathtt{i}}] = \mathrm{cha}[\pi_{\mathtt{o}}]$ because the prefixes can communicate. Assume without loss of generality that $\pi_{\mathtt{i}} = c(x)$ and $\pi_{\mathtt{o}} = \overline{c}\langle b\rangle$, then

$$\begin{aligned}\Lambda^R(\alpha) &= \langle \mathrm{dat}[\pi_{\mathtt{i}}] \mapsto \{b\}\rangle, \text{ def'n of } \Lambda^R \\ &= \langle x \mapsto \{b\}\rangle \qquad (5.2)\end{aligned}$$

The process $C\|P$ can reduce by communication in only three ways:

1. The communication occurring between two prefixes in $C$
2. The communication occurring between two prefixes in $P$
3. The communication occurring in an interaction between $C$ and $P$

The result is now proved by case analysis on these possibilities:

*Case.* ($\pi_{\mathtt{i}} \notin P \wedge \pi_{\mathtt{o}} \notin P$) In this case $\mathcal{X}_P(\alpha) = \emptyset$ and $c$ and $b$ are either free in $P$ or do not appear in $P$ as in the previous case. The name $x$ cannot appear in $P$ because it is bound in $C$. The results of the analysis for this action is computed as:

$$\text{lfp}(\Phi_{\{\}}) = (\bot_R, \bot_K, \bot_E)$$

Since $x$ is not in $P$, $\Lambda(\alpha)\nabla P = \bot$, where $\bot$ again denotes the function returning the empty set for each input. In this case the analysis result is certainly greater than the actual interaction because $\bot \subseteq \bot_R$ by definition, yielding the $R$ component of the result. The other two components are likewise empty under the $\Lambda$ mapping because the entire prefix interaction is in $C$.

*Case.* ($\pi_{\mathtt{i}} \in P \wedge \pi_{\mathtt{o}} \in P$) If both prefixes are in $P$, then $\Lambda^R(\alpha)\nabla P = \Lambda(\alpha)$ and $\mathcal{X}_P(\alpha) = \mathcal{X}(\alpha) = \{\pi_{\mathtt{i}}, \pi_{\mathtt{o}}\}$. By the definition of $\Phi$

$$\text{lfp}(\Phi_{\mathcal{X}(\alpha)}) = (\bot_R \sqcup \langle x \mapsto \{b\}\rangle, \bot_K \sqcup \langle c \mapsto \{b\}\rangle, \bot_E)$$

and since $\langle x \mapsto \{b\}\rangle$ is included in the analysis solution, the result for the $R$ component follows from equation 5.2. The results for the other components follow because

$$\begin{aligned} \Lambda^K_C(\alpha) &= \langle \text{cha}[\pi_{\mathtt{i}}] \mapsto \{b\}\rangle, \text{ def'n of } \Lambda^K_C \\ &= \langle c \mapsto \{b\}\rangle \\ \Lambda^E_C(\alpha) &= \bot_E \text{ because both prefixes are in } P \end{aligned} \tag{5.3}$$

and the analysis solution is consistent with both of them.

*Case.* ($\pi_{\mathtt{i}} \notin P \wedge \pi_{\mathtt{o}} \in P$) This case involves an interaction between a prefix in $P$ wanting to transmit to the environment $C$. Since the input prefix $c(x) \notin P$, then $\mathcal{X}_P(\alpha) = \{\overline{c}\langle b\rangle\}$, and the name $c$ must be free in $P$ (because it is known to $C$ and there are no preceding actions that got it there because all the substitutions are empty). Furthermore, $x \notin \text{n}(P)$ because it is bound by $c(x) \in C$, and $b$ is not bound by input

because its substitution is empty. The analysis result is given by

$$\mathrm{lfp}(\Phi_{\{\overline{c}\langle b\rangle\}}) = (\bot_R, \bot_K \sqcup \langle c \mapsto \{b\}\rangle, \bot_E \cup \{b\})$$

which yields the result for the $R$ component when compared with $\Lambda^R(\alpha)\nabla P = \bot$ (because $x \notin P$). The symbol $\bot$ here denotes the function that returns the empty set for every input. The other two components follow trivially.

*Case.* ($\pi_{\mathtt{i}} \in P \wedge \pi_{\mathtt{o}} \notin P$) This case handles the situation whereby the environment transmits a value into $P$. In this case $\mathcal{X}_P(\alpha) = \{c(x)\}$ and $c$ is again free in $P$. Furthermore, $b$ is either free in $P$ or does not appear in $P$ because there is no previous action that could have extruded $b$ and substituted it for some bound name in $C$ (since the substitutions in the action are empty). The result of the analysis for this action is computed as:

$$\mathrm{lfp}(\Phi_{\{c(x)\}}) = (\bot_R \sqcup \langle x \mapsto \bot_E\rangle, \bot_K, \bot_E)$$

In contrast, $\Lambda^R(\alpha)$ maps $x$ to $\{b\}$. However, since $\bot_E = \mathrm{fn}(P) \cup \xi_P$, if $b \in \mathrm{fn}(P)$ then it is included in $\bot_E$, and if $b \notin \mathrm{n}(P)$ then it is included in $\xi_P$ by definition and thus also in $\bot_E$, yielding the result for the $R$ component of the claim. The $K$ component is computed as:

$$\Lambda^K_C(\alpha) = \bot_K$$

because the output prefix is in $C$, and the $E$ component follows trivially.

□

The result is now extended to arbitrary actions $\alpha$ in a trace by reasoning on the structure of the dependency tree of $\alpha$:

**Lemma 5.1.4.** *Given a $\pi$-calculus process $P$, let $\alpha = (\pi_{\mathtt{i}}, \pi_{\mathtt{o}})$ be an action in some trace $tr$ in $\mathbf{H}[C\|P]$, and let $(R, K, E) = \mathrm{lfp}(\Phi_{\mathcal{X}_P(\alpha)})$ in $\mathcal{L}_{\mathcal{E}}$, then*

$$(\Lambda^R(\alpha)\nabla P, \Lambda^K_C(\alpha), \Lambda^E_C(\alpha)) \sqsubseteq (R, K, E)$$

*Proof.* By induction on the dependency tree of $\alpha$, which admits induction because it is finite. The base case is where the tree is just the action $\alpha$, i.e., $\alpha$ has no children, and its proof is given by the previous lemma. The case where the $\alpha$ is not a leaf implies that there are three actions $\alpha^c_{\mathtt{i}}, \alpha^c_{\mathtt{o}}, \alpha^d_{\mathtt{o}}$ preceding $\alpha$ (i.e., that are $\alpha$'s children in its dependency tree) that caused each name to differ from its original expression, and these are of the form:

$$\tilde{\alpha} = (\tilde{\pi}_{\mathtt{i}}, \tilde{\pi}_{\mathtt{o}})$$

for $\tilde{\alpha} \in \{\alpha^c_{\mathtt{i}}, \alpha^c_{\mathtt{o}}, \alpha^d_{\mathtt{o}}\}$. The cases where some of the substitutions are empty and some are not are special cases of this one. The induction hypothesis states the following:

$$(\Lambda^R(\tilde{\alpha})\nabla P, \Lambda^K_C(\tilde{\alpha}), \Lambda^E_C(\tilde{\alpha})) \sqsubseteq \tilde{\lambda}$$

where $\tilde{\lambda} = (R', K', E') = \mathrm{lfp}(\Phi_{\mathcal{X}_P(\tilde{\alpha})})$ for each $\tilde{\alpha}$ a child of $\alpha$ in the dependency tree. The induction step is now proved by generalizing the case analysis on $\alpha$ shown in the proof of the previous lemma. In the following, the triple $(\widetilde{R}, \widetilde{K}, \widetilde{E})$ is defined as the join of the solutions computed inductively for each subtree:

$$(\widetilde{R}, \widetilde{K}, \widetilde{E}) = \bigsqcup_{\tilde{\lambda}} \tilde{\lambda}$$

i.e., the join of the sub-solutions for each sub-tree. As the generalization follows a similar pattern in each case, only one of these is explicitly shown. Suppose without loss of generality that $\alpha$ is the following action:

$$\alpha = (a(w), \overline{a}\langle b \rangle)$$

and that

$$\begin{aligned} \sigma(a(w)) &= x(w) \\ \sigma(\overline{a}\langle b \rangle) &= \overline{y}\langle z \rangle \end{aligned}$$

are the original forms of the prefixes in the action.

*Case.* ($\pi_{\mathtt{i}} \in P \wedge \pi_{\mathtt{o}} \notin P$) In this case the environment is transmitting a value to $P$. It is thus known that $\mathcal{X}_P(\alpha) = \{x(w)\} \cup \bigcup_{\tilde{\alpha}} \mathcal{X}_P(\tilde{\alpha})$, i.e., the output prefix is not in the set of prefixes in the dependency tree of $\alpha$ that also occur in $P$. Here $\Lambda^R(\alpha)\nabla P = \langle w \mapsto \{b\}\rangle$ because $w \in \mathrm{n}(P)$. Therefore, it must be shown that the $R$ component of the analysis includes $b$ in the set $R(w)$. The analysis solution can be computed by adding the contribution of $x(w)$ to the intermediate solutions for the subtrees:

$$\mathrm{lfp}(\Phi_{\mathcal{X}_P(\alpha)}) = (\widetilde{R} \sqcup \langle w \mapsto \widetilde{K}(\widetilde{R}(x)) \cup \widetilde{E}\rangle, \widetilde{K}, \widetilde{E})$$

This follows from the definition of $\Phi^{\mathcal{I}}$. Thus it is sufficient to show that $b \in (\widetilde{K}(\widetilde{R}(x)) \cup \widetilde{E})$ to prove the $R$ component of the result. Multiple applications of the induction hypothesis can now be used to track the flow of $b$ through the process in order to establish the result. Because of the datum of the output prefix in $\alpha$ was originally $z$ (and is now $b$), it is known that $\alpha^d_{\mathtt{o}}$ must have been an action that transmitted $b$ and stored the transmitted name into $z$. However, since $\pi_{\mathtt{o}}$ is in the environment $C$, the action $\alpha^d_{\mathtt{o}}$ must at least have its input prefix in the environment (because the name $z$ must have been bound by an input prefix preceding $\pi_{\mathtt{o}}$). This leaves the sub-cases where $\alpha^d_{\mathtt{o}}$'s output prefix is in $P$ or in $C$. In the former case, the $E^d_{\mathtt{o}}$ of the induction hypothesis applies to $\alpha^d_{\mathtt{o}}$ and yields that $b \in E^d_{\mathtt{o}}$. In the second case, the name $b$ must either be free in $P$ or not appear in $P$ (because it is known to $C$ at this point), which also implies that it is in $E^d_{\mathtt{o}}$ because it is in $\bot_E$. Since $\widetilde{E}$ is just the join of $E'$ for each $\tilde{\alpha} \in \{\alpha^c_{\mathtt{i}}, \alpha^c_{\mathtt{o}}, \alpha^d_{\mathtt{o}}\}$, it is also true that $b \in \widetilde{E}$ yielding $R$ component of the result. The other cases are demonstrated in a similar fashion.

□

This lemma essentially states that the operation of the analysis on all of the prefixes in $P$ that a particular action depends on is sufficient to compute a correct result for that action. The full correctness of the dataflow analysis can now be shown

by the following theorem:

**Theorem 5.1.5.** *Given a $\pi$-calculus process $P$, and any context $C$, let $\lambda = (\widehat{R}, \widehat{K}, \widehat{E}) = (\Lambda^R(\mathbf{H}[C\|P]), \Lambda^K_C(\mathbf{H}[C\|P]), \Lambda^E_C(\mathbf{H}[C\|P]))$ and let $(R, K, E) = \mathrm{lfp}(\Phi_{[\![P]\!]})$ in $\mathcal{L}_\mathcal{E}$ with $[\![P]\!] = [\![P]\!]_\emptyset$, then*

$$(\widehat{R}\nabla P, \widehat{K}, \widehat{E}) \sqsubseteq (R, K, E)$$

*Proof.* Suppose some name $c \in (\widehat{R}\nabla P)(x)$ for some name $x \in \mathrm{n}(P)$. This implies the existence of some action $\alpha$ in some trace $tr$ in $\mathbf{H}[C\|P]$ such that $(\Lambda^R(\alpha))(x) = \{c\}$ because $\Lambda$ compositionally combines its result over each action. By lemma 5.1.4 the following statement is true:

$$\Lambda(\alpha)\nabla P \sqsubseteq \widetilde{R} \tag{5.4}$$

where $(\widetilde{R}, \widetilde{K}, \widetilde{E}) = \mathrm{lfp}(\Phi_{\mathcal{X}_P(\alpha)})$. By definition, $\mathcal{X}_P(\alpha) \subseteq [\![P]\!]$, and by the definition of $\Phi$, this yields

$$\mathrm{lfp}(\Phi_{\mathcal{X}_P(\alpha)}) \sqsubseteq \mathrm{lfp}(\Phi_{[\![P]\!]}) \tag{5.5}$$

because adding prefixes to the representation can only increase the final solution. Finally, making the appropriate substitutions into equations 5.4 and 5.5 yields

$$\Lambda^R(\alpha)\nabla P \sqsubseteq \widetilde{R} \sqsubseteq R$$

and since $x \in \mathrm{n}(P)$ this implies that $c \in R(x)$ yielding the result. The other two components are proved in the same way. □

Notice that this proof shows that the $R$ component of the solution is the only one that is independent of the context with which $P$ is placed in parallel composition for the reasons stated above.

## 5.2 Considering Matching

Having developed a correct context independent dataflow analysis of the $\pi$-calculus, it now makes sense to begin discussions of making it more accurate. This is essentially done by refining the representation of processes to contain more information about the original $\pi$-calculus process being analyzed, and modifying the abstract execution function to properly exploit this information. In order to illustrate this idea, a simple refinement of the representation above is presented that is equivalent to the implicit representation used by Bodei *et al.* in [BDNN01b]. Informally, the feature exploited is based on the observation that an input or output prefix $\pi$ in process $P$ can't communicate unless all of the matches that precede it can successfully pass.

### 5.2.1 Refined Process Representation

Recall from section 4.1.1 that a $\pi$-calculus process $P$ was represented by its set of observable prefixes $\mathcal{X}_P$ under the representation function $[\![\cdot]\!]_\emptyset$. This representation makes it impossible to determine whether a prefix is blocked from communicating due to a match. For instance, if $P \triangleq (\nu a)(\nu b)([a = b].(\overline{c}\langle a\rangle \| c(x)))$, then the function $\Phi_{[\![P]\!]}$ would compute that $a \in R(x)$ due to the fact that $a$ can be transmitted over $c$ and received by the prefix $c(x)$. However, this is clearly impossible because the action is blocked by the match prefix $[a = b]$, which can never pass because both $a$ and $b$ are bound to the process. In order to detect such a blockage, it is necessary to refine the naive process representation $[\![\cdot]\!]_\emptyset$ as follows: rather than storing prefixes individually, they are paired with the set of match prefixes that precede them.

Recall that $\mathcal{M}_P$ denotes the set of match prefixes of process $P$. Just as was done for input and output prefixes (which were formalized as pairs in $\mathcal{I} = (\mathcal{N} \times \mathcal{N})$ and $\mathcal{O} = \langle \mathcal{N} \times \mathcal{N} \rangle$ respectively), a match prefix $[x = y]$ is formalized as a pair $[x, y] \in [\mathcal{N} \times \mathcal{N}]$ (note the square brackets for distinction), and the set of all such

prefixes is denoted by $\mathcal{M}$. As with the other prefix sets, the universal sets of all possible prefixes (excluding $\tau$) is denoted $\Pi = \mathcal{X} \cup \mathcal{M}$.

Subsets of $\mathcal{M}$ are paired with each observable prefix to form the set $\mathcal{X} \times 2^{\mathcal{M}}$, and the refined representation of a process $P$ will be a set of such pairs. This refined representation is computed as follows:

**Definition 5.2.1.**

$$\begin{aligned}
[\![P \| Q]\!]^C_{\mathcal{M}} = [\![P + Q]\!]^C_{\mathcal{M}} = &\quad [\![P]\!]^C_{\mathcal{M}} \cup [\![Q]\!]^C_{\mathcal{M}} && \forall P, Q \\
[\![(\nu x)P]\!]^C_{\mathcal{M}} = [\![\tau.P]\!]^C_{\mathcal{M}} = [\![!P]\!]^C_{\mathcal{M}} = &\quad [\![P]\!]^C_{\mathcal{M}} && \forall P\ \forall x \in \mathcal{N} \\
[\![\pi.P]\!]^C_{\mathcal{M}} = &\quad \{(\pi, C)\} \cup [\![P]\!]^C_{\mathcal{M}} && \forall P\ \forall \pi \in \mathcal{X}_P \\
[\![[x = y].P]\!]^C_{\mathcal{M}} = &\quad [\![P]\!]^{C \cup \{[x,y]\}}_{\mathcal{M}} && \forall P
\end{aligned}$$

where $C \subseteq \mathcal{M}$. The final match-sensitive representation function $[\![\cdot]\!]_{\mathcal{M}}$ is computed by $[\![P]\!]_{\mathcal{M}} = [\![P]\!]^{\emptyset}_{\mathcal{M}}$ for any $P$.

Notice that this definition is identical to the inductive definition of $[\![\cdot]\!]_{\emptyset}$ given by proposition 4.1.1 with the exception of the match prefix set $C$. This set carries with it the set of all matches encountered while descending the parse tree of the process. Observe that each prefix is added to the set in the rule for $[x = y].P$, and the entire set $C$ is paired with each input/output prefix when encountered in the rule for $\pi.P$ (for $\pi \in \mathcal{X}_P$). For illustrative purposes, we contrast the two representations by using the process $P$ from example 5.0.2:

**Example 5.2.1.** Consider the following process:

$$P \triangleq a(x).b(y).[x = y].(\nu c)(\nu d)(\overline{d}\langle c\rangle \| d(z).\overline{a}\langle z\rangle)$$

The representations $[\![P]\!]_{\emptyset}$ and $[\![P]\!]_{\mathcal{M}}$ are computed as follows:

$$\begin{aligned}
[\![P]\!]_{\emptyset} &= \{a(x), b(y), \overline{d}\langle c\rangle, d(z), \overline{a}\langle z\rangle\} \\
[\![P]\!]_{\mathcal{M}} &= \{(a(x), \emptyset), (b(y), \emptyset), \\
&\qquad (\overline{d}\langle c\rangle, \{[x = y]\}), (d(z), \{[x = y]\}), (\overline{a}\langle z\rangle, \{[x = y]\})\}
\end{aligned}$$

The cardinalities of the two sets are equal, but the prefixes that precede the test $[x = y]$ (i.e., $a(x)$ and $b(y)$) are paired with the empty set while those that follow it are paired with the singleton set containing the match prefix.

This representation can be exploited to improve the accuracy of the analysis, and this is done in the following section.

### 5.2.2 Abstract Execution

The modification of the iterative execution function must preserve the context-independence of the $\Phi$ function, thus the solution computed will still be a triple in $\mathcal{L}_{\mathcal{E}}$. In fact, only a simple modification is required to account for the blockage of prefixes due to an unpassable match.

Now, given a set $S \subseteq (\mathcal{X} \times 2^{\mathcal{M}})$, the new abstract execution function $\widetilde{\Phi}_S : \mathcal{L}_{\mathcal{E}} \to \mathcal{L}_{\mathcal{E}}$ is defined as follows:

**Definition 5.2.2.**

$$\widetilde{\Phi}_S(\lambda) = \bigsqcup_{(\pi,M)\in S} \begin{cases} \Phi_\pi(\lambda) & \text{if } \forall [x,y] \in M : (R(x) \cap R(y)) \neq \emptyset \\ \bot_{\mathcal{E}} & \text{otherwise} \end{cases}$$

Where $(\pi, M)$ is a prefix $\pi$ matched with a set of matches $M$. Since the restricting condition

$$\forall [x,y] \in M : (R(x) \cap R(y)) \neq \emptyset$$

is checkable in polynomial time for any finite process (because $|M|$ is at worst linear in the size of $P$), the entire flow analysis remains computable in polynomial time.

This function is identical to $\Phi_{[\![P]\!]}$ for $[\![P]\!] = [\![P]\!]_\emptyset$ if $[\![P]\!]_\emptyset$ is viewed as pairings in the set $(\mathcal{X} \times 2^\emptyset)$ rather than individual prefixes (this explains the motivation of the $\emptyset$ subscript in the notation). Formally, if $[\![\widetilde{P}]\!]$ is taken as the set $[\![P]\!]_\emptyset$ viewed as such pairs, and $[\![P]\!] = [\![P]\!]_\emptyset$, then it is obvious that:

$$\widetilde{\Phi}_{[\![\widetilde{P}]\!]} = \Phi_{[\![P]\!]}$$

The $\widetilde{\Phi}$ function simply has the effect of applying the $\Phi$ function only to those prefixes whose preceding matches can pass (as given by the condition $R(x) \cap R(y) \neq \emptyset$), other prefixes have no effect on the result (reflected by having them contribute $\bot_{\mathcal{E}}$ to the solution). This is exactly the same property shared by the constraint based model of Bodei *et al.* [BDNN01b] (see appendix A.1), where constraints are only imposed for such prefixes. It would be expected that the least solution satisfying Bodei *et al.*'s constraint system should be equivalent to the solution computed by $\widetilde{\Phi}_S$. However, Bodei *et al.*'s solution does not consider the environment of the analyzed process, and thus is only correct for closed processes. Thus, as expected, the two solutions are only equivalent for closed processes:

**Proposition 5.2.1.** *Given a closed $\pi$-calculus process $P$, let $(\rho, \kappa)$ be the least solution to the constraints generated for $P$ by the judgement in table A.1, and let $\lambda = (R, K, E) = \mathrm{lfp}(\widetilde{\Phi}_{[\![P]\!]})$ with $[\![P]\!] = [\![P]\!]_{\mathcal{M}}$. Then*

$$(\rho, \kappa) = (R, K)$$

*with equality taken in $\mathcal{L}_{\emptyset}$.*

*Proof.* By the definition of $\widetilde{\Phi}$, $(R, K) = \widetilde{\Phi}_{[\![P]\!]} = \Phi_T$ for $T$ the following set:

$$\{\pi \in [\![P]\!]_{\emptyset} \mid (\pi, M) \in [\![P]\!]_{\mathcal{M}} \wedge \forall [x, y] \in M : R(x) \cap R(y) \neq \emptyset\}$$

In words, $\widetilde{\Phi}$ acts as $\Phi$ operating on only the prefixes whose matches can pass. Lemma 5.1.2 yields that $\Phi_T$ behaves as $\Omega_T$ because $P$ is a closed process. Theorem 4.1.3 yields that the solution $\Phi_T$ satisfies the same constraints as $(\rho, \kappa)$ for prefixes in $T$, and by the observations made about the rule for matching in table A.1 (namely that constraints are not imposed for prefixes not in $T$) it is known that no additional constraints are satisfied yielding the result. □

Note that this proposition's statement closely follows the statement of theorem 4.1.3, except that it establishes the *equivalence* of the two approaches. This result establishes the correctness of the solution for closed processes. The next section proves the correctness of the solution in any context.

### 5.2.3 Correctness

The main advantage of the proof technique used in section 5.1.2 is that it makes similar context-independent proofs for subsequent analyses rather easy. This is because the accuracy of these analyses are achieved by restricting the contributions of prefixes in the analyzed process $P$ based on a particular condition. In the case of the above analysis based on preceding matches, the dataflow contributions of prefixes that succeed matches $[x = y]$ that cannot pass (as given by the condition $R(x) \cap R(y) = \emptyset$) are added to the final solution. The correctness of the algorithm can thus be proved with a technique similar to that used to prove proposition 5.2.1 for closed processes. The context-independent result is proved because the basic correctness theorem (i.e., theorem 5.1.5) for such an analysis has been established.

Specifically, the correctness of $\widetilde{\Phi}$ is proved by appealing to the correctness of $\Phi$ and then showing that the condition $R(x) \cap R(y) \neq \emptyset$ doesn't exclude any prefixes that could communicate. First, a new operator on processes is introduced:

**Definition 5.2.3.** Given a process $P$, and a set of match prefixes $M \subseteq \mathcal{M}_P$, define the process $P \Uparrow M$ as the process $P$ "pruned" at the prefixes in $M$, i.e., $P \Uparrow M$ is the same process $P$ with all of the subprocesses that succeed every match prefix in $M$ removed.

Formally, the process $P \Uparrow M$ is defined by applying the operator $\Uparrow$ to each subprocess in the structural definition of $P$ with the exception of the following cases:

$$[x = y].P \Uparrow M \quad = \quad \mathbf{0} \text{ if } [x, y] \in M$$

$$\mathbf{0} \Uparrow M \;=\; \mathbf{0}$$

Otherwise, the operator is just pushed into the subprocesses in the definition until one of the above rules is reached. The correctness of the algorithm is proved only for the $R$ component, as it is the one that is correct independent of context, and thus presents the most interesting result without cluttering the section with similar proofs for the other components. The correctness is given by the following theorem:

**Theorem 5.2.2.** *Given a $\pi$-calculus process $P$, and any context $C$, let $\widehat{R} = \Lambda^R(\mathbf{H}[C\|P])$, and let $(R, K, E) = \mathrm{lfp}(\widetilde{\Phi}_{[\![P]\!]})$ in $\mathcal{L}_\mathcal{E}$ with $[\![P]\!] = [\![P]\!]_\mathcal{M}$, then*

$$\widehat{R}\nabla P \sqsubseteq R$$

*Proof.* Let $\overline{M} \subseteq \mathcal{M}_P$ be the set of match prefixes $[x = y]$ of $P$ such that $R(x) \cap R(y) = \emptyset$, i.e., $\overline{M}$ is the set of match prefixes in $P$ that did not pass in any iteration of the analysis. By the definition of $\widetilde{\Phi}$ it is known that:

$$(R, K, E) = \mathrm{lfp}(\widetilde{\Phi}_{[\![P]\!]}) = \mathrm{lfp}(\widetilde{\Phi}_{[\![P \Uparrow \overline{M}]\!]}) \tag{5.6}$$

i.e., the solution does not change if the prefixes whose preceding matches don't pass are excluded from the representation because they only contribute $\bot_\mathcal{E}$ to the final solution. Furthermore, also by the definition of $\widetilde{\Phi}$:

$$\mathrm{lfp}(\widetilde{\Phi}_{[\![P \Uparrow \overline{M}]\!]}) = \mathrm{lfp}(\Phi_{[\![P \Uparrow \overline{M}]\!]_\emptyset}) \tag{5.7}$$

i.e., the operation of the new function on those prefixes whose preceding matches pass is identical to the operation of the previous function (that did not consider blocking matches) on those prefixes. This claim is justified because it is exactly how the $\widetilde{\Phi}$ function was defined in definition 5.2.2. Letting $(\widetilde{R}, \widetilde{K}, \widetilde{E}) = \mathrm{lfp}(\Phi_{[\![P \Uparrow \overline{M}]\!]_\emptyset})$, i.e., the solution to the previous dataflow analysis on the abbreviated process, the correctness of the $\Phi$ function, as given by theorem 5.1.5 implies the following:

$$\Lambda^R[\mathbf{H}[C\|(P \Uparrow \overline{M})]]\nabla P \sqsubseteq \widetilde{R} \tag{5.8}$$

By equations 5.6 and 5.7 it is also known that $\widetilde{R}(x) \cap \widetilde{R}(y) = \emptyset$ for each match prefix $[x, y] \in \overline{M}$. Therefore, this result also applies to $\Lambda^R[\mathbf{H}[C \| (P \Uparrow \overline{M})]]$ by equation 5.8, which in turn implies that the match prefix $[x = y]$ does not pass in any trace of $C \| P$, and thus none of the potential subsequent actions appear in the traces, i.e.,

$$\mathbf{H}[C \| P] = \mathbf{H}[C \| (P \Uparrow \overline{M})] \tag{5.9}$$

Combining equations 5.6, 5.7, 5.8, and 5.9 yields

$$\widehat{R} \;=\; \Lambda^R[\mathbf{H}[C \| P]] \nabla P \;=\; \Lambda^R[\mathbf{H}[C \| (P \Uparrow \overline{M})]] \nabla P \;\sqsubseteq\; \widetilde{R} \;=\; R$$

Which is the claimed result. □

The technique presented in the latter part of this chapter essentially gives a partial treatment of the $\pi$-calculus sequencing operator (".") by testing for unpassable match prefixes. This approach can be extended to apply for input and output prefixes as well. This is formally developed in the first sections of the following chapter. The latter part of the chapter also adds the consideration of the parallel composition ("‖") and choice ("+") operators in order to provide the most accurate dataflow analysis of the $\pi$-calculus to date, in that the pairs of prefixes that can potentially communicate are further limited by the analysis.

# Chapter 6

# Sequencing and Sub-Process Structure

The analysis developed in the latter half of the previous chapter provides a context-independent extension of the analysis of Bodei *et al.* from [BDNN01b]. The analysis extended the one presented in chapter 4 by pairing each prefix with the set of match prefixes that precede it, and not applying the contribution of the potential communications of that prefix unless it has been determined that the matches that precede it can pass. The latter determination was made by testing the condition $(R(x) \cap R(y) \neq \emptyset)$ for each match prefix $[x, y]$.

This suggests a natural extension: pairing each prefix with *all* prefixes that precede it in the process, and only applying the contribution of the prefix if it has been determined that all of its predecessors can communicate (if they are observable prefixes) or pass (if they are match prefixes).

The first section of the this chapter develops the analysis described here, and the second section provides another extension that further improves the accuracy of the solution. The last section gives some insight as to how the analysis could be implemented in order to improve its efficiency.

## 6.1 Blocking on All Prefixes

An analysis that considers blocked input and output prefixes requires the development of the following elements:

- A modification of the representation to allow for all preceding prefixes to be paired with each prefix
- An appropriate condition that can be used to test if a prefix has communicated according to the current solution

The first is handled by the next subsection, and the second is integrated into the definition of the abstract execution function.

### 6.1.1 Abstract Representation

Recall that $\Pi$ was defined in chapter 3 as the set of all prefixes (excluding $\tau$), i.e., $\Pi = \mathcal{X} \cup \mathcal{M}$. This motivates the alteration of the representation $[\![P]\!]_{\mathcal{M}}$ of process $P$ as a set of pairings in $(\mathcal{X} \times 2^{\mathcal{M}})$ to a representation $[\![P]\!]_{\Pi}$ comprising a set of pairings in $(\mathcal{X} \times 2^{\Pi})$ in the obvious way:

**Definition 6.1.1.**

$$\begin{aligned} [\![P \| Q]\!]^C_\Pi = [\![P + Q]\!]^C_\Pi = &\ [\![P]\!]^C_\Pi \cup [\![Q]\!]^C_\Pi && \forall P, Q \\ [\![(\nu x)P]\!]^C_\Pi = [\![\tau.P]\!]^C_\Pi = [\![!P]\!]^C_\Pi = &\ [\![P]\!]^C_\Pi && \forall P\ \forall x \in \mathcal{N} \\ [\![\pi.P]\!]^C_\Pi = &\ \{(\pi, C)\} \cup [\![P]\!]^{C \cup \{\pi\}}_\Pi && \forall P\ \forall \pi \in \Pi \end{aligned}$$

where $C \subseteq \Pi$. The final prefix-sensitive representation function $[\![\cdot]\!]_\Pi$ is denoted $[\![P]\!]_\Pi = [\![P]\!]^\emptyset_\Pi$ for any $P$.

Recall that assumption 2 states that the "prefixes" in the representation are actually referring to occurrences of prefixes to keep these distinct, thus each prefix $\pi'$ in the predecessor set $C$ of a pair $\eta = (\pi, C)$ is associated bijectively to another pair

$\eta' = (\pi', C')$ in the representation. Since the association is one-to-one, the prefix $\pi$ is used interchangeably with the pair $\eta$. When distinction is required, projections are defined on a pair $\eta = (\pi, C)$ as follows:

**Definition 6.1.2.** Given a pair $\eta = (\pi, C) \in (\Pi \times 2^{\Pi})$ the projection returning the prefix of the pair is denoted $\mathbf{this}[\eta] = \pi$, and access to the set of predecessors is provided by the function $\mathbf{pset}[\eta] = C$.

Due to the correspondence, it should cause no confusion to talk about a prefix pair $\eta = (\pi, C)$ as being an input, output, or match prefix. Thus the notation $\eta \in \mathcal{I}$, $\eta \in \mathcal{O}$, etc. is used to mean $\mathbf{this}[\eta] \in \mathcal{I}$, etc..

The next section develops an execution function that exploits the additional information included in this representation.

### 6.1.2 Abstract Execution

As observed above, a necessary requirement of a correct analysis that exploits the potential of blocking prefixes is the development of a condition that correctly captures the ability of a prefix to communicate. The latter half of the previous chapter already provides such a condition for match prefixes. However, observable prefixes in $\mathcal{X}$ differ from matches in that they need to be paired with another prefix in order to communicate: an input prefix must have an output prefix to communicate with and vice versa. As such, a notion of which prefixes are able to synchronize must be defined. In order to do so, a predicate is defined to define the conditions under which the action of a prefix may occur.

**Definition 6.1.3.** Given an intermediate solution $\lambda = (\widehat{R}, \widehat{K}, \widehat{E}) \in \mathcal{L}_{\mathcal{E}}$, an observable prefix $\pi \in \mathcal{X}$ is *environment enabled* (denoted $\lambda \vdash_E \pi$) with respect to $\lambda$ if, and only if

$$\widehat{E} \cap \widehat{R}(\mathrm{cha}[\pi]) \neq \emptyset$$

This definition is extended to prefix/predecessor pairs $\eta \in \Pi \times 2^{\Pi}$ in the obvious way, i.e.,

$$\lambda \vdash_E \eta \Leftrightarrow \lambda \vdash_E \mathbf{this}[\eta]$$

This definition captures the ability of a prefix to communicate with the environment. If $\pi$ is an input prefix, then $\lambda \vdash_E \pi$ denotes that $\pi$ could receive a communication from the environment, and similarly that $\pi$ could transmit to the environment if $\pi$ is an output prefix.

Similarly, the ability of a match prefix to pass, according to the condition used in the previous chapter, is encapsulated into the enabling relation by the following definition:

**Definition 6.1.4.** Given an intermediate solution $\lambda = (\widehat{R}, \widehat{K}, \widehat{E}) \in \mathcal{L}_{\mathcal{E}}$, a match prefix $[x, y] \in \mathcal{M}$ is *match enabled* (denoted $\lambda \vdash_M [x = y]$) with respect to $\lambda$ if, and only if

$$\widehat{R}(x) \cap \widehat{R}(y) \neq \emptyset$$

This definition is extended to prefix/predecessor pairs $\eta \in \Pi \times 2^{\Pi}$ in the obvious way, i.e.,

$$\lambda \vdash_M \eta \Leftrightarrow \lambda \vdash_M \mathbf{this}[\eta]$$

Finally, the ability of an input/output prefix pair to communicate is encapsulated into an enabling relation as follows:

**Definition 6.1.5.** Given an intermediate solution $\lambda = (\widehat{R}, \widehat{K}, \widehat{E}) \in \mathcal{L}_{\mathcal{E}}$, a prefix pair $(\pi_{\mathtt{i}}, \pi_{\mathtt{o}}) \in \mathcal{I} \times \mathcal{O}$ is *communication enabled* (denoted $\lambda \vdash_C (\pi_{\mathtt{i}}, \pi_{\mathtt{o}})$) with respect to $\lambda$ if, and only if

$$\widehat{R}(\mathrm{cha}[\pi_{\mathtt{i}}]) \cap \widehat{R}(\mathrm{cha}[\pi_{\mathtt{o}}]) \neq \emptyset$$

This definition is extended to prefix/predecessor pairs $\eta \in \Pi \times 2^{\Pi}$ in the obvious way, i.e.,

$$\lambda \vdash_C (\eta, \eta') \Leftrightarrow \lambda \vdash_C (\mathbf{this}[\eta], \mathbf{this}[\eta'])$$

This captures the notion that the input prefix can communicate with the output prefix if both their channels could be the same name (as given by the $\widehat{R}$ function). It is obvious that each of these predicates are checkable in polynomial time given a prefix. The ability of an individual prefix in a process $P$ to communicate can now be recursively encoded by the following definition:

**Definition 6.1.6.** Given an intermediate solution $\lambda = (\widehat{R}, \widehat{K}, \widehat{E}) \in \mathcal{L}_{\mathcal{E}}$, and a set $S$ of prefixes paired with their predecessors, a pair $\eta \in S$ is *enabled* (without qualification) with respect to $\lambda$ and $S$ (denoted $\lambda; S \vdash_\infty \eta$) if, and only if

$$\lambda; S \vdash_\infty \mathbf{pset}[\eta] \wedge \begin{cases} \lambda \vdash_M \eta & \text{if } \eta \in \mathcal{M} \\ \lambda \vdash_E \eta \quad \vee \quad \Big(\exists \eta' \in S : \ \lambda \vdash_C (\eta, \eta') \wedge \lambda; S \vdash_\infty \mathbf{pset}[\eta']\Big) & \text{if } \eta \in \mathcal{X} \end{cases}$$

The notation $\lambda; S \vdash_\infty \mathbf{pset}[\eta]$ extends the relation to sets in the obvious way, i.e.,

$$\lambda; S \vdash_\infty \mathbf{pset}[\eta] \Leftrightarrow \forall \eta' \in \mathbf{pset}[\eta] : \lambda; S \vdash_\infty \eta'$$

This definition says that a prefix is enabled if all of its predecessors are enabled, and:

- it is match-enabled if it is a match prefix, or
- it is either environment enabled, or it is communication enabled with some other prefix whose predecessors are enabled if it is an observable prefix

Unfortunately, directly computing the predicate $\lambda; S \vdash_\infty \eta$ for some $\eta$ could potentially lead to infinite regress due to the recursion on the $\vdash_\infty$ predicate. However,

thanks to the iterative nature of the fixed point computation, an equivalent decidable condition can be computed by keeping track of which prefixes have potentially communicated at each step of the abstract execution function.

This is done by expanding the solution $\mathcal{L}_{\mathcal{E}}$ space with a function $\mathcal{C} : \Pi_{\perp} \to \mathbb{B}$, with $\Pi_{\perp}$ the lifted set of prefixes and $\mathbb{B}$ the two point lattice $\mathbb{ff} \sqsubseteq \mathbb{tt}$. The new solution space is denoted $\mathcal{L}_{\Psi}$ and is comprised of all 4-tuples $(R, K, E, \mathcal{C})$ with $\mathcal{C}$ initialized by $\mathcal{C}[\perp] = \mathbb{tt}$ and $\mathbb{ff}$ for every other value. The initialized function is the bottom element $\perp_{\mathcal{C}}$ in the lattice $\mathcal{L}_{\mathcal{C}} \subseteq [\Pi_{\perp} \Rightarrow \mathbb{B}]$ comprised of all the functions above $\perp_{\mathcal{C}}$.

Furthermore, define the *immediate predecessor* of a pair $\eta = (\pi, C)$ in a set $S$ of such pairs as the unique prefix $\eta' \in C$ such that

$$\mathbf{pset}[\eta] \setminus \mathbf{pset}[\eta'] = \{\eta'\}$$

The immediate predecessor of $\eta$ is denoted by the function $\rho[\cdot] : \Pi \to \Pi_{\perp}$ such that

$$(\mathbf{pset}[\eta] = \emptyset) \Rightarrow (\rho[\eta] = \perp)$$

This property, combined with the fact that $\mathcal{C}$ is initalized to return $\mathbb{tt}$ on an input of $\perp$, will have the effect of computing that the "predecessors" of a prefix with no predecessors have communicated. The checkable condition used in the abstract execution function is now given in terms of the function $\mathcal{C}$ as follows:

**Definition 6.1.7.** Given an intermediate solution $\lambda = (\widehat{R}, \widehat{K}, \widehat{E}, \widehat{\mathcal{C}})$, and a set $S$ of prefixes paired with their predecessors, a pair $\eta \in S$ is *enabled* (without qualification) with respect to $\lambda$ and $S$ (denoted $\lambda; S \vdash \eta$) if, and only if

$$\widehat{\mathcal{C}}[\rho[\eta]] \wedge \begin{cases} \lambda' \vdash_M \eta & \text{if } \eta \in \mathcal{M} \\ \lambda' \vdash_E \eta \quad \vee \quad \Big(\exists \eta' \in S : \ \lambda' \vdash_C (\eta, \eta') \wedge \widehat{\mathcal{C}}[\rho[\eta']]\Big) & \text{if } \eta \in \mathcal{X} \end{cases}$$

Where $\lambda'$ denotes the triple $(\widehat{R}, \widehat{K}, \widehat{E})$.

This is identical to the definition of enabling given in definition 6.1.6 except that the recursion is removed by appealing to the current values of the given function $\widehat{\mathcal{C}}$, thus given a function $\widehat{\mathcal{C}}$ this predicate can be checked in polynomial time. The abstract execution function is now defined to ensure that the function $\widehat{\mathcal{C}}$ correctly satisfies the definition of enabling given in definition 6.1.6.

The analysis function is defined over the lattice $\mathcal{L}_\Psi = \mathcal{L}_\mathcal{E} \times \mathcal{L}_\mathcal{C}$, i.e., it updates the set of prefixes that are known to communicate in each iteration (given by the function $\mathcal{C} \in \mathcal{L}_\mathcal{C}$).

Given $\lambda = (R, K, E, \mathcal{C}) \in \mathcal{L}_\Psi$, and a set $S$ of pairs $\eta = (\pi, C)$, the execution function $\Psi : \mathcal{L}_\Psi \to \mathcal{L}_\Psi$ is defined as follows:

**Definition 6.1.8.**

$$\Psi_S(\lambda) = \bigsqcup_{\eta \in S} \begin{cases} (\Phi_\eta(R, K, E), \langle \eta \mapsto \mathrm{tt} \rangle) & \text{if } \lambda; S \vdash \eta \\ \bot_\mathcal{E} & \text{otherwise} \end{cases}$$

The Greek letter $\Psi$ is used as a mnemonic for PSequential. Note the similarity between this definition and the definition of $\widetilde{\Phi}$, which only added the contribution of a prefix if its preceding matches passed. In this case, the contribution of a prefix according to the flow insensitive function $\Phi$ is only added if it is enabled according to the condition $\lambda; S \vdash \eta$. Furthermore, each such prefix whose contribution is added is marked as having communicated by having its image under $\mathcal{C}$ set to true.

The final solution $\lambda_{sol}$ is computed in the usual way by iterating this function from bottom, given a $\pi$-calculus process $P$, the final solution is computed by

$$\lambda_{sol} = \mathrm{lfp}(\Psi_{[\![P]\!]})$$

with $[\![P]\!] = [\![P]\!]_\Pi$. Since the relation $\vdash$ can be checked in polynomial time, the analysis function remains computable in polynomial time as well.

### 6.1.3 Correctness

The proof of correctness is going to be quite similar to the proof of the correctness of the $\widetilde{\Phi}$ function that blocked on matches only, because the same ideas are used here on all prefixes. The main difference is the condition used to determine if a prefix is blocked, therefore the main result required to prove correctness is that the updates of the $\mathcal{C}$ component of the solution correctly emulate the desired condition given by the predicate $\vdash_\infty$ at each iteration of the computation. That proof itself requires a sub-lemma given below:

**Lemma 6.1.1.**

$$\lambda; S \vdash_\infty \eta \;\; \Rightarrow \;\; \lambda'; S \vdash_\infty \eta \quad \textit{if} \;\; \lambda \sqsubseteq \lambda'$$
$$\lambda; S \vdash \eta \;\; \Rightarrow \;\; \lambda'; S \vdash \eta \quad \textit{if} \;\; \lambda \sqsubseteq \lambda'$$

*Proof Sketch.* The result states that if a prefix is enabled under $\vdash_\infty$ in the context of a solution $\lambda$, then it is also enabled in the context of a larger solution. The result is quite evident from the definition, because the checked conditions depend on the intersections of sets defined by the functions $R$ and $E$ (which only get larger as $\lambda$ gets larger), and by checking the truth values of assigned to each prefix by the $\mathcal{C}$ component of the solution (in the case of the $\vdash$ relation) which can only set more prefixes to true as the solution gets larger. □

The next lemma shows that the $\vdash$ and $\vdash_\infty$ relations are in fact equivalent for solutions generated by iterating $\Psi$ on a process.

**Lemma 6.1.2.** *Let $\lambda = (R, K, E, \mathcal{C}) = \Psi^{(k)}_{[\![P]\!]}(\bot_\Psi)$ for some $\pi$-calculus process $P$, with $[\![P]\!] = [\![P]\!]_\Pi$, and some integer $k$, then*

$$\lambda; S \vdash \eta \;\; \Leftrightarrow \;\; (R, K, E); S \vdash_\infty \eta$$

*Proof.* Both directions are proved by induction on $k$, but only the $\Rightarrow$ direction is explicitly shown as the proof of the reverse direction is practically identical. By induction on $k$. When $k = 0$, $\lambda = \perp_\Psi$ implying that $\mathbf{pset}[\eta] = \emptyset$ by the definition of $\perp_\mathcal{C}$, and that the recursive condition $(R, K, E); S \vdash_\infty \emptyset$ is met because the set is empty. If $\eta$ is match-enabled or environment enabled, then certainly $(R, K, E); S \vdash_\infty \eta$ because these conditions are identically checked in the $\vdash_\infty$ definition. If there exists an $\eta' \in S$ such that $\mathcal{C}[\rho[\eta']] = \Lightning$, this implies that $\mathbf{pset}[\eta'] = \emptyset$ as well because $\mathcal{C} = \perp_\mathcal{C}$. By the same reasoning as above, this implies that the recursive condition $(R, K, E); S \vdash_\infty \mathbf{pset}[\eta']$ is trivially true because the set $\mathbf{pset}[\eta']$ is empty. For the induction step, consider the tuple $\lambda_k = (R_k, K_k, E_k, \mathcal{C}_k) = \Psi^{(k)}_{[\![P]\!]}(\perp_\Psi)$, and the application of another iteration of the function to generate $\lambda_{k+1} = (R_{k+1}, K_{k+1}, E_{k+1}, \mathcal{C}_{k+1}) = \Psi_{[\![P]\!]}(\lambda_k)$. Now consider a prefix $\eta \in S$ such that $\lambda_{k+1}; S \vdash \eta$. By the definition of $\vdash$, this implies that $\mathcal{C}_{k+1}[\rho[\eta]] = \Lightning$. By the definition of $\Psi$, this then implies that $\rho[\eta]$ was enabled in the previous iteration, i.e., $\lambda_k; S \vdash \rho[\eta]$. By the induction hypothesis it is then known that $(R_k, K_k, E_k); S \vdash_\infty \rho[\eta]$, which implies that each of the predecessors of $\rho[\eta]$ satisfies the predicate $\vdash_\infty$ under the solution $(R_k, K_k, E_k)$, and by lemma 6.1.1, each of these prefixes also satisfies $\vdash_\infty$ under the solution $(R_{k+1}, K_{k+1}, E_{k+1})$ because $\lambda_k \sqsubseteq \lambda_{k+1}$ by the monotonicity of $\Psi$. Formally, it is known that:

$$\forall \eta' \in \mathbf{pset}[\eta] : \lambda_{k+1}; S \vdash_\infty \eta' \tag{6.1}$$

and by the assumption that $\lambda_{k+1}; S \vdash \eta$ it is known that $\eta$ is either communication enabled, environment enabled, or match enabled, and combined with equation 6.1, this implies

$$\lambda_{k+1}; S \vdash_\infty \eta$$

by the definition of $\vdash_\infty$, which completes the induction. □

It is now possible to prove the correctness of the $\Psi$ function in much the same way as the correctness of the $\widetilde{\Phi}$ function was proved:

**Theorem 6.1.3.** *Given a $\pi$-calculus process $P$, and any context $C$, let $\widehat{R} = \Lambda^R(\mathbf{H}[C\|P])$, and let $\lambda = (R, K, E, \mathcal{C}) = \mathrm{lfp}(\Psi_{[\![P]\!]})$ in $\mathcal{L}_\Psi$ with $[\![P]\!] = [\![P]\!]_\Pi$, then*

$$\widehat{R} \nabla P \sqsubseteq R$$

*Proof.* Let $\overline{S} \subseteq \Pi_P$ be the set of prefixes $\eta$ of $P$ such that $\mathcal{C}[\eta] = \mathrm{ff}$, i.e., $\overline{S}$ is the set of prefixes in $P$ that were computed as not having communicated, because $\mathcal{C}[\eta] = \mathrm{ff}$ at the fixed point implies that $\lambda; [\![P]\!] \nvdash \eta$, and consequently that $\lambda; S \nvdash_\infty \eta$ by the previous lemma. thus the contributions of these prefixes were not added to the final solution. Thus, by the definition of $\Psi$:

$$(R, K, E, \mathcal{C}) = \mathrm{lfp}(\Psi_{[\![P]\!]}) = \mathrm{lfp}(\Psi_{[\![P \Uparrow \overline{S}]\!]}) \tag{6.2}$$

i.e., the solution does not change if the prefixes whose preceding prefixes don't communicate are excluded from the representation because they only contribute $\bot_\Psi$ to the final solution. Furthermore, also by the definition of $\Psi$:

$$\mathrm{lfp}(\Psi_{[\![P \Uparrow \overline{S}]\!]}) = (\mathrm{lfp}(\Phi_{[\![P \Uparrow \overline{S}]\!]_\emptyset}), \mathcal{C}) \tag{6.3}$$

i.e., the operation of the new function on those prefixes whose preceding prefixes do not communicate is identical to the operation of the non-sequential function $\Phi$ on those prefixes. This claim is justified because it is exactly how the $\Psi$ function was defined in definition 6.1.8. Letting $(\widetilde{R}, \widetilde{K}, \widetilde{E}) = \mathrm{lfp}(\Phi_{[\![P \Uparrow \overline{S}]\!]_\emptyset})$, i.e., the solution to the previous dataflow analysis on the abbreviated process, the correctness of the $\Phi$ function, as given by theorem 5.1.5 implies the following:

$$\Lambda^R[\mathbf{H}[C\|(P \Uparrow \overline{S})]] \nabla P \sqsubseteq \widetilde{R} \tag{6.4}$$

By equations 5.6 and 5.7 it is also known that $(\widetilde{R}, \widetilde{K}, \widetilde{E}, \mathcal{C}); [\![P]\!] \nvdash \eta$ for each prefix $\eta \in \overline{S}$. Therefore, this result also applies to $\Lambda^R[\mathbf{H}[C\|(P \Uparrow \overline{S})]]$ by equation 5.8, which

in turn implies that the prefix prefix $\eta$ does not communicate in any trace of $C\|P$, and thus none of the potential subsequent actions appear in the traces, i.e.,

$$\mathbf{H}[C\|P] = \mathbf{H}[C\|(P \Uparrow \overline{S})] \tag{6.5}$$

Combining equations 6.2, 6.3, 6.4, and 6.5 yields

$$\widehat{R} \;=\; \Lambda^R[\mathbf{H}[C\|P]]\nabla P \;=\; \Lambda^R[\mathbf{H}[C\|(P \Uparrow \overline{S})]]\nabla P \;\sqsubseteq\; \widetilde{R} \;=\; R$$

Which is the claimed result. □

## 6.2 Sub-Process Structure

The last enhancement of the analysis developed in this thesis is based on a simple observation. Up to this point, it has been assumed that any input prefix in a process could potentially communicate with any output prefix in the process, so long as they could potentially share a channel and that they are not blocked by preceding prefixes. However, this is not necessarily the case, the two prefixes in the process $P \triangleq (\nu a)(\nu b)(a(x) + \overline{a}\langle b\rangle)$ satisfy both of these conditions, but it is still impossible for them to communicate because they are on opposite sides of a choice operator. The previous analysis would compute that $R(x) = K(a) = \{b\}$ even though $P$ is unable to reduce in any context.

The main observations used to restrict the potentially communicating prefixes of a process are then as follows:

- Prefixes on opposite sides of a choice operator can not communicate
- Prefixes on opposite sides of a parallel composition operator can communicate
- All prefixes in a replicated process can communicate with one another

This section focuses on formalizing these intuitive observations by developing yet another process representation that is able to detect valid prefix pairs, and then again modifying the previous abstract execution function $\Psi$ to exploit the representation with the goal of further improving the accuracy of the analysis.

### 6.2.1 Abstract Representation

In order to track communications in the process, prefixes that can potentially communicate with one another must be defined. In the flow-insensitive analyses previously defined, where composition and choice operators are not taken into account, it is assumed that every input prefix can communicate with every output prefix and vice versa. The enhanced analysis of Bodei *et al.* in [BDPZ03] accounts for the fact that prefixes on the opposite sides of a choice operator cannot communicate, but doesn't account for the fact that a prefix cannot communicate until all of it's predecessors have had a chance to do so. The representation here combines the ideas in [BDPZ03] (summarized in appendix A.2) with the sequential representation defined in the previous section in order to improve the accuracy of the analysis.

The first step towards such a representation is the generation of the parse tree of a process $P$. This is explicitly defined in order to illustrate how the sequencing of prefixes is taken into account. Each pairing of a prefix $\pi$ with its predecessor set is represented by a node in the parse tree, as well as each of the process operators "$\|$" and "+". The definition of a parse tree consists of a set of nodes defined by the following grammar:

$$\begin{aligned}
\text{Node} &::= \text{PrefixNode} \mid \text{OpNode} \mid \text{PrefixNodeSet} \\
\text{PrefixNode} &::= (\pi, S) \\
\text{PrefixNodeSet} &::= \{\text{PrefixNode}_1, \ldots, \text{PrefixNode}_n\} \\
\text{OpNode} &::= \| \mid +
\end{aligned}$$

A node is either the pairing of a prefix $\pi$ with its predecessor set $S$, a process operator, or a set of prefix pairs. The latter is used to represent a replicated process, where flow information is lost due to the structural congruence $!P \equiv P \| !P$ which allows any prefixes in $P$ to appear on opposite sides of a composition operator. The functions **this**[PrefixNode] and **pset**[PrefixNode] provide access to the elements of a PrefixNode as before.

Process trees are then defined as follows:

$$\begin{aligned} \text{ProcTree} &::= \varepsilon \mid \langle \text{Node}, \text{Children} \rangle \\ \text{Children} &::= \{\text{ProcTree}_1, \ldots, \text{ProcTree}_n\} \end{aligned}$$

A tree is either an empty tree $\varepsilon$, or the pairing of a Node (accessed by the projection **root**[ProcTree]) and a pair of ProcTrees. The function $\mathbf{child}_i$[ProcTree] returns the $i^{th}$ ProcTree in the set of Children. A PrefixNode $\eta$ is defined to be in a ProcTree $T$ (denoted $\eta \in T$) if $\eta$ is a node of $T$, or $\eta \in S$ where $S$ is a PrefixNodeSet that is a node of $T$. The notation $\Pi_P$ for the set of prefixes of a process $P$ is extended to $\Pi_T$ denoting the set of PrefixNodes in a ProcTree $T$, and similar extensions $\mathcal{X}_T$, $\mathcal{O}_T$, $\mathcal{I}_T$, and $\mathcal{M}_T$ are also defined.

Given these definitions, the abstract representation of a process $P$ is defined by a function $[\![P]\!]_{\mathcal{T}}^{C}$ taking a $\pi$-calculus process $P$ and returning a ProcTree (the element $C$ is used to keep track of the predecessors as before). The tree representation can now be defined inductively on the structure of $P$

**Definition 6.2.1.**

$$\begin{aligned}
[\![\mathbf{0}]\!]^C_{\mathcal{T}} &= \varepsilon \\
[\![P \| Q]\!]^C_{\mathcal{T}} &= \langle \|, \{[\![P]\!]^C_{\mathcal{T}}, [\![Q]\!]^C_{\mathcal{T}}\}\rangle \\
[\![P + Q]\!]^C_{\mathcal{T}} &= \langle +, \{[\![P]\!]^C_{\mathcal{T}}, [\![Q]\!]^C_{\mathcal{T}}\}\rangle \\
[\![(\nu x)P]\!]^C_{\mathcal{T}} = [\![\tau.P]\!]^C_{\mathcal{T}} &= [\![P]\!]^C_{\mathcal{T}} \\
[\![\pi.P]\!]^C_{\mathcal{T}} &= \langle(\pi, C), \{[\![P]\!]^{\{\pi\}\cup C}_{\mathcal{T}}\}\rangle \\
[\![!P]\!]^C_{\mathcal{T}} &= [\![P]\!]^C_{\Pi}
\end{aligned}$$

The final abstract representation of a process $[\![P]\!]_{\mathcal{T}}$ is computed by $[\![P]\!]^{\emptyset}_{\mathcal{T}}$.

An example of the construction is presented in figure 6.2.1 on the process from example 5.1.1:

$$P \triangleq a(x).(\nu b)(\nu c)((\overline{b}\langle a\rangle.\overline{x}\langle x\rangle.b(y).\overline{y}\langle c\rangle + !\overline{b}\langle d\rangle.\overline{a}\langle c\rangle) \parallel b(z).\overline{b}\langle z\rangle) \parallel d(w)$$

This representation simply generates the parse tree of $P$, with the exception of a replicated process $!P$, which is interpreted as the set of its prefixes because of the reasons outlined above. The next section defines a function that explicitly computes the pairs of structurally compatible prefixes, and then refines the execution function to exploit the representation.

### 6.2.2 Abstract Execution

The computation of the valid prefix pairs is done by a function $\mathcal{E}()$ : ProcTree $\rightarrow$ (PrefixNode $\times$ PrefixNode) generating the pairs of prefixes that can *structurally* communicate:

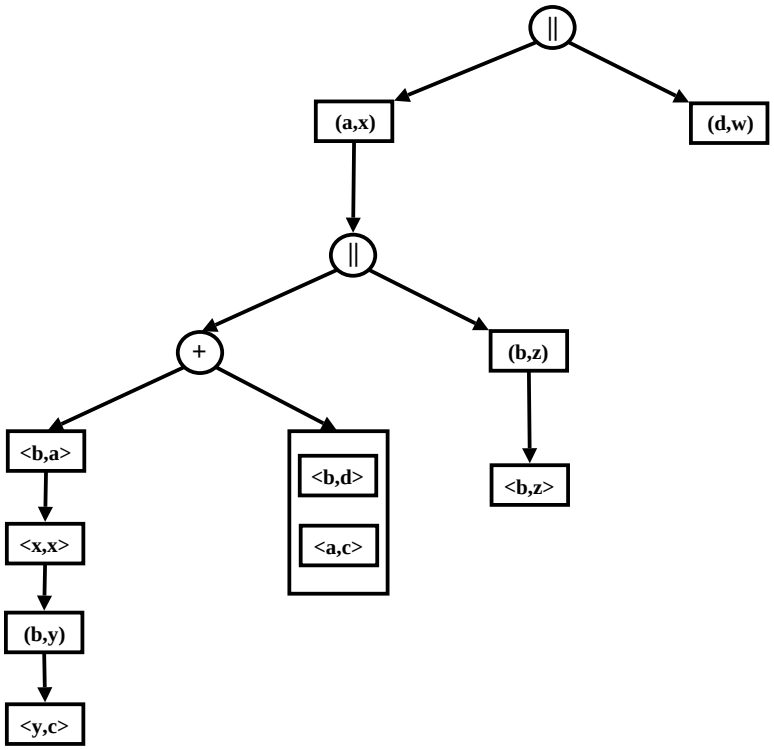



Figure 6.1: The tree $[\![P]\!]_{\mathcal{T}}$ for the process $P$ in example 5.1.1

**Definition 6.2.2.** Given a $\pi$-calculus process $P$, define the set of structurally communication enabled prefixes as $\mathcal{E}([\![P]\!]_{\mathcal{T}})$, with $\mathcal{E}()$ the following function:

$$\begin{aligned}
\mathcal{E}(\varepsilon) &= \emptyset \\
\mathcal{E}(\langle \eta, \{T\} \rangle) &= \mathcal{E}(T) \\
\mathcal{E}(\langle +, \{T_1, T_2\} \rangle) &= \textstyle\bigcup_i \mathcal{E}(T_i) \\
\mathcal{E}(\langle \|, \{T_1, T_2\} \rangle) &= \Delta(\{T_1, T_2\}) \cup \textstyle\bigcup_i \mathcal{E}(T_i) \\
\mathcal{E}(\langle S, \emptyset \rangle) &= (\mathcal{O}_S) \times (\mathcal{I}_S)
\end{aligned}$$

where $\eta$ is a PrefixNode, $S$ is a PrefixNodeSet, $T$ is a ProcTree, $\{T_1, T_2\}$ is a ProcTreeSet, and $\Delta$ is defined below.

The letter $\mathcal{E}()$ is used as a mnemonic for "edges", indicating that "communication edges" between PrefixNodes are being generated . The first equation in definition 6.2.2 handles the empty tree. The second says that a PrefixNode does not generate any new pairs. The third handles the case that prefixes on opposite sides of a choice operator can't communicate by returning all the pairs in the summed children without generating any pairs *between* the prefixes in them. The fourth equation handles processes in parallel composition by generating the same set of pairs within each child tree *in addition to* the set of pairs $\Delta(\{T_1, T_2\})$, where this set comprises each input (output) PrefixNode in each tree $T_i$ paired with each output (input) PrefixNode in the other tree in the set, i.e.,

$$\Delta(\{T_1, T_2\}) = \Big( (\mathcal{I}_{T_1} \times \mathcal{O}_{T_2}) \cup (\mathcal{I}_{T_2} \times \mathcal{O}_{T_1}) \Big)$$

The final equation in definition 6.2.2 handles the case of a replicated sub-process by generating the set of all input/output pairs in the given NodeSet $S$.

Now, in order to show how to exploit this in the abstract execution function, recall that the definition of an enabled prefix $\eta$ (definition 6.1.7) in the context of a solution

$\lambda = (\widehat{R}, \widehat{K}, \widehat{E}, \widehat{\mathcal{C}})$ and a prefix set $S$ was as follows:

$$\widehat{\mathcal{C}}[\rho[\eta]] \wedge \begin{cases} \lambda' \vdash_M \eta & \text{if } \eta \in \mathcal{M} \\ \lambda' \vdash_E \eta \quad \vee \quad \Big(\exists \eta' \in S : \ \lambda' \vdash_C (\eta, \eta') \wedge \widehat{\mathcal{C}}[\rho[\eta']]\Big) & \text{if } \eta \in \mathcal{X} \end{cases}$$

Where $\lambda'$ denotes the triple $(\widehat{R}, \widehat{K}, \widehat{E})$. Specifically, observe that the communication condition (in the case of an observable prefix) requires the existence of some other prefix $\eta' \in S$ with which $\eta$ can communicate. Thus, the only modification that needs to be made to the abstract execution function in order to further restrict its solution is to restrict this existential quantification to prefixes that are paired with $\eta$ in $\mathcal{E}([\![P]\!]_{\mathcal{T}})$. Specifically, the notion of enabling with respect to a solution $\lambda$ and a prefix set $S$ is defined below:

**Definition 6.2.3.** Given an intermediate solution $\lambda = (\widehat{R}, \widehat{K}, \widehat{E}, \widehat{\mathcal{C}})$, and a ProcTree $T$, a prefix $\eta \in T$ is *enabled* with respect to $\lambda$ and $T$ (denoted $\lambda; T \vdash \eta$) if, and only if

$$\widehat{\mathcal{C}}[\rho[\eta]] \wedge \begin{cases} \lambda' \vdash_M \eta & \text{if } \eta \in \mathcal{M} \\ \lambda' \vdash_E \eta \quad \vee \quad \Big(\exists (\eta, \eta') \in \mathcal{E}(T) : \ \lambda' \vdash_C (\eta, \eta') \wedge \widehat{\mathcal{C}}[\rho[\eta']]\Big) & \text{if } \eta \in \mathcal{X} \end{cases}$$

Where $\lambda'$ denotes the triple $(\widehat{R}, \widehat{K}, \widehat{E})$.

The abstract execution function is then modified to use this definition of enabling rather than the previous one, i.e.,

**Definition 6.2.4.**

$$\Psi_T(\lambda) = \bigsqcup_{\eta \in T} \begin{cases} (\Phi_\eta(R, K, E), \langle \eta \mapsto \mathrm{tt} \rangle) & \text{if } \lambda; T \vdash \eta \\ \bot_{\mathcal{E}} & \text{otherwise} \end{cases}$$

Note that $\Psi$ now operates on a tree $T$ rather than a set $S$. The final solution is computed in the standard iterative way, except that now $[\![P]\!] = [\![P]\!]_{\mathcal{T}}$. It is easy to

see that the correctness of this analysis follows from the correctness of the previous one so long as the set $\mathcal{E}(\llbracket P \rrbracket_{\mathcal{T}})$ does not exclude any prefix pairs that could actually communicate during the reduction of the process. This result is proved in the following proposition:

**Proposition 6.2.1.** *Given a $\pi$-calculus processes $P$ and $C$, and any action $\alpha = ((\pi_{\mathtt{i}}, \sigma^c_{\mathtt{i}}, \bot_\sigma), (\pi_{\mathtt{o}}, \sigma^c_{\mathtt{o}}, \sigma^d_{\mathtt{o}}))$ in $\mathbf{H}[C \| P]$ such that $\pi_{\mathtt{i}} \in \mathcal{X}_P$ and $\pi_{\mathtt{o}} \in \mathcal{X}_P$, then $(\pi_{\mathtt{i}}, \pi_{\mathtt{o}}) \in \mathcal{E}(\llbracket P \rrbracket_{\mathcal{T}})$*

*Proof.* The claim states that if a pair of prefixes that are in the prefix set of $P$ communicate at some point in some trace of $C \| P$, then that pair must be included in $\mathcal{E}(\llbracket P \rrbracket_{\mathcal{T}})$. If there exists some subprocess $!P'$ in $P$ such that both $\pi_{\mathtt{i}} \in \mathcal{X}_{P'}$ and $\pi_{\mathtt{o}} \in \mathcal{X}_{P'}$, then the result is achieved since all input/output pairs in $P'$ are included in $\mathcal{E}(\llbracket P \rrbracket_{\mathcal{T}})$ by definition. If this is not the case, then by the definition of prefix communication there must exist subprocesses $P_1$ and $P_2$ of $P$ such that $P_1 \| P_2$ is a subprocess of $P$, and (without loss of generality) that $\pi_{\mathtt{i}} \in \mathcal{X}_{P_1}$ and $\pi_{\mathtt{o}} \in \mathcal{X}_{P_2}$. By definition

$$\mathcal{E}(\llbracket P_1 \| P_2 \rrbracket_{\mathcal{T}}) \subseteq \mathcal{E}(\llbracket P \rrbracket_{\mathcal{T}})$$

because $P_1 \| P_2$ is a subprocess of $P$, and by applying the rule for generating $\mathcal{E}(\llbracket P_1 \| P_2 \rrbracket_{\mathcal{T}})$, it is known that

$$\Delta(\{\llbracket P_1 \rrbracket_{\mathcal{T}}, \llbracket P_2 \rrbracket_{\mathcal{T}}\}) \subseteq \mathcal{E}(\llbracket P_1 \| P_2 \rrbracket_{\mathcal{T}})$$

and by the assumption that $\pi_{\mathtt{i}} \in \mathcal{X}_{P_1}$ and $\pi_{\mathtt{o}} \in \mathcal{X}_{P_2}$, and the definition of $\Delta$ it is known that:

$$(\pi_{\mathtt{i}}, \pi_{\mathtt{o}}) \in \Delta(\{\llbracket P_1 \rrbracket_{\mathcal{T}}, \llbracket P_2 \rrbracket_{\mathcal{T}}\})$$

and finally, the transitivity of set inclusion yields the desired result that $(\pi_{\mathtt{i}}, \pi_{\mathtt{o}}) \in \mathcal{E}(\llbracket P \rrbracket_{\mathcal{T}})$. □

This completes the formal definitions of the analyses of the $\pi$-calculus defined in this thesis. The effect of these enhancements will now be explored through a series of security examples in the next chapter. These examples will highlight some of the abilities, as well as the limitations of, the dataflow analyses developed in the previous chapters.

# Chapter 7
# Application: Examples in Security

Program analyses can be used to determine various properties of programs at compile time. In the context of the $\pi$-calculus, programs define a concurrent system in which agents can transmit data and channel capabilities to one another. The program analyses presented in the previous chapters track the dataflow through such a system, as described by a $\pi$-calculus process. One potential use for the information gleaned in the analysis solution is to check whether the system in question satisfies a particular security property. This chapter provides a discussion-style treatment of how security properties can be checked using various degrees of accuracy in the dataflow analysis solution.

Specifically, the accuracy of a dataflow analysis will affect the corresponding security analysis as follows: inaccuracy in the dataflow analysis will lead to false positives in the security analysis. An inaccurate analysis will always provide a correct analysis – i.e. insecure processes will always be rejected – but the security analysis may also reject secure processes. Improving the accuracy of the dataflow analysis will reject fewer secure processes.

The first section describes how some basic security properties can be modelled

by the $\pi$-calculus, and presents examples that illustrate some of the abilities and limitations of the analyses presented. The second section informally introduces an extension of the $\pi$-calculus that adds basic cryptographic primitives to the calculus, and discusses how the flow analyses presented in this thesis could simply be extended to reason about properties of cryptographic protocols.

## 7.1 Confidentiality In The Π-Calculus

The term "secrecy" has numerous meanings in the program security literature, one usage of the term is as a synonym for confidentiality where it is assured that a private piece of data is not leaked to an agent that is not authorized to see it. This property can naturally be expressed in the $\pi$-calculus, where names can be viewed as data. Given a $\pi$-calculus process $P$, the names of $P$ can be partitioned into two sets $\mathcal{S}$ and $\mathcal{P}$ depicting the names that must be kept secret, and those that can be made public respectively. It is then simple to check the result $(R, K, E)$ of the dataflow analysis to ensure confidentiality properties such as:

- Secret names are not transmitted over public channels (i.e., $K(\mathcal{P}) \cap \mathcal{S} = \emptyset$)
- Private names are not leaked to the environment (i.e., $\mathcal{S} \cap E = \emptyset$)
- Publicly observable input variables don't receive secret names (i.e., this implies checking the condition $R(\mathcal{P}) \cap \mathcal{S} = \emptyset$)

### 7.1.1 Example: Context Dependency

As a preliminary example, consider the process $P$ from example 5.0.2:

$$P \triangleq a(x).b(y).[x = y].(\nu c)(\nu d)(\overline{d}\langle c\rangle \| d(z).\overline{a}\langle z\rangle)$$

and consider the behaviour of the match-sequential analysis $\widetilde{\Phi}_{[\![P]\!]_{\mathcal{M}}}$ defined in section 5.2. By proposition 5.2.1, closing this process and computing its solution under $\widetilde{\Phi}$ will produce the same result as the analysis of Bodei *et al.* from [BDNN01b], i.e., if $(\rho, \kappa)$ is the solution of the latter analysis on $P$, and $(R, K, E) = \text{lfp}(\widetilde{\Phi}_{[\![(\nu a)(\nu b)P]\!]_{\mathcal{M}}})$, where $(\nu a)(\nu b)P$ is the closed version of $P$, then

$$(\rho, \kappa) = (R, K)$$

In fact, this analysis produces $\bot_{\mathcal{E}}$ as a solution because both functions conclude that the match prefix $[x = y]$ can't pass. Bodei *et al.*'s analysis does this on the open process because it does not consider the environment, and the $\widetilde{\Phi}$ function arrives at this conclusion because it was given a closed version of the process. However, note that Bodei *et al.*'s analysis computed this result on the open version of the process (i.e., without binding $a$ and $b$). However, computing the $\text{lfp}(\widetilde{\Phi}_{[\![P]\!]_{\mathcal{M}}})$ function on the open version of the process produces the following solution:

$$\begin{array}{rclrcl} R(x) & = & \{a, b, c, \xi_P\} & K(a) & = & \{c\} \\ R(y) & = & \{a, b, c, \xi_P\} & K(b) & = & \{\} \\ R(z) & = & \{c\} & K(c) & = & \{\} \\ E & = & \{a, b, c, \xi_P\} & K(d) & = & \{c\} \end{array}$$

If the set of private names is set in the natural way to be the bound names of $P$, i.e., $\mathcal{S} = \text{bn}(P) = \{x, y, z, d, c\}$, the solution above shows that the private name $c$ was leaked to the environment because $E \cap \mathcal{S} = \{c\}$, and also that it was sent over the public channel $a$ because $a \in \mathcal{P}$ and $K(a) \cap \mathcal{S} = \{c\}$. This example shows how the analysis presented in section 5.2 accounts for the reduction sequence presented in example 5.0.2 when $P$ was run concurrently with a context $C$ that received the leaked name.

### 7.1.2 Example: Wide-Mouthed Frog

Recall the introductory example from chapter 1:

$$\begin{array}{lll} \text{Message 1} & A \rightarrow S: & A, \{B, K_{AB}\}_{K_{AS}} \quad \text{on } c_S \\ \text{Message 2} & S \rightarrow B: & \{A, K_{AB}\}_{K_{SB}} \quad \text{on } c_B \\ \text{Message 3} & A \rightarrow B: & A, \{M\}_{K_{AB}} \quad \text{on } c_B \end{array}$$

This protocol models the behaviour of two agents $A$ and $B$ that wish to transmit an encrypted message, but lack the shared keys required to do so. In order to establish such a key, the agents communicate with a server $S$ that they both trust (and with which they each share a key), and use a series of communications to perform the key exchange. The agent $A$ first generates a key $K_{AB}$ that it would like to use as a shared key to send a message to $B$. It sends this key to $S$, paired with the name of the agent with which it would like to communicate, and encrypted with the key $K_{AS}$ that it shares with the server. The server in turn sends an encrypted message to $B$ with the key $K_{AB}$ paired with the name of the agent that would like to communicate with it. After these steps are completed, $A$ and $B$ both know the key $K_{AB}$, so $A$ can encrypt the message with it and send it to $B$.

Due to the lack of encryption and decryption primitives in the $\pi$-calculus, this protocol must be modelled under the following assumptions (see [BDNN01b]):

1. Knowledge of a key is modelled by knowledge of a channel
2. Encryption is modelled by transmission over a channel
3. Decryption is modelled by reception on a channel

Under these assumptions, the Wide Mouthed Frog protocol can be expressed as follows in the $\pi$-calculus (adapted from [BDNN01b]):

$$A \;=\; (\nu M)(\nu c_{AB})\overline{c_{AS}}\langle c_{AB}\rangle.\overline{c_{AB}}\langle M\rangle$$

$$\begin{aligned} S &= c_{AS}(x).\overline{c_{SB}}\langle x\rangle \\ B &= c_{SB}(y).y(z) \\ P &= (\nu c_{AS})(\nu c_{SB})(A\|S\|B) \end{aligned}$$

The process $A$ has the action of sending the "key" $c_{AB}$ to $S$ "encrypted" with $c_{AS}$, and then waiting until it is able to send $M$ to $B$ encrypted with $c_{AB}$. The server $S$ decrypts a message from $A$ and puts it into $x$, and then sends the received value (which will be the key $c_{AB}$) to $B$ encrypted with $c_{SB}$. The process $B$ waits for a value from $S$ which it decrypts with $c_{SB}$, and then waits to decrypt a message with the key it received.

As the process $P$ is closed, and contains no prefixes that are unable to communicate, the basic analysis $\Omega_{[\![P]\!]_\emptyset}$ is sufficient to get the dataflow information, as all of the subsequent analyses would produce the same result. This analysis produces the following solution when iterated to a fixed point:

$$\begin{aligned} R(x) &= \{c_{AB}\} & K(c_{AB}) &= \{M\} \\ R(y) &= \{c_{AB}\} & K(c_{AS}) &= \{c_{AB}\} \\ R(z) &= \{M\} & K(c_{SB}) &= \{c_{AB}\} \\ & & K(M) &= \{\} \end{aligned}$$

This solution indicates the exact transmissions and encryptions of $M$ and $c_{AB}$ through the $K$ function. It also indicates that the secrecy of the message $M$ hinges on the secrecy of the key $c_{AB}$ that it is encrypted with (because $K(c_{AB}) = \{M\}$), and that the secrecy of $c_{AB}$ hinges on the secrecy of the keys ($c_{AS}$ and $c_{SB}$) that it is encrypted with. The latter keys are not transmitted in any way according to the analysis, and thus the secrecy of the message $M$ is guaranteed.

Now observe what happens if the protocol were modified to have $A$ explicitly leak the key $c_{AB}$ after transmitting the message, i.e., if $A$ were modified to the following

process:

$$A' = (\nu M)(\nu c_{AB})\overline{c_{AS}}\langle c_{AB}\rangle.\overline{c_{AB}}\langle M\rangle.\overline{c_{AC}}\langle c_{AB}\rangle$$

which is the same as $A$ suffixed by the transmission of the private key $c_{AB}$ encrypted with some key $c_{AC}$ and sent to some outside party $C$. In this case, the result of the analysis would compute that $K(c_{AC}) = \{c_{AB}\}$, and since $c_{AC}$ would be free in $P$ (and hence a public name by the above convention), then the secrecy of the message $M$ would be compromised because $K(\mathcal{P}) \cap \mathcal{S}$ is non-empty.

## 7.2 Integrity

Another oft-used term in the security literature is that of data integrity, and one potential definition of it is the property that an agent's data does not get corrupted by another agent. There are various potential versions of a data integrity property, but in this section we examine the particular one in which data integrity is secured by only allowing the owner of a piece of data to access it. The simple example below will also serve to illustrate how more accurate analyses than the basic flow insensitive one may be needed to properly do this.

### 7.2.1 Example: Accuracy

The different levels of accuracy of the analyses developed in the previous chapters are shown by analyzing and refining a simple example. Consider the following family of processes:

$$\begin{aligned} C_i &\triangleq \overline{in_S}\langle k_i\rangle.out_S(x_i) \\ S &\triangleq in_S(y).\left(\sum_{i\in\{1,\ldots,N\}} [y = k_i].\overline{out_S}\langle D_i\rangle\right) \end{aligned}$$

The process $S$ defines a server with two communication channels $in_S$ and $out_S$ for input and output respectively. Upon receiving a name on its input channel, the server compares it with one of $N$ possible names $k_1, \ldots, k_n$, and if any are successful transmits a name $D_i$ on its output channel depending on which match succeeded. This process can be viewed as modelling a password access database, where the $k_i$'s represent passwords, and the $D_i$'s represent a private data record that is accessible upon presentation of the name $k_i$.

The processes $C_i$ then represent potential clients of the database. Each client transmits its password $k_i$ on the server's input channel, and then waits for a reply on the server's output channel.

It is interesting to see in some detail how the analyses presented in the previous chapters behave on configurations of these processes. First, the case of a single client interacting with the server in isolation is considered. Let $j$ be an integer such that $1 \leq j \leq N$, then consider the process

$$P \triangleq (\nu\ in_S)(\nu\ out_S)(\nu\ k_1, \ldots, k_N)(\nu\ D_1, \ldots, D_N)(S \| C_j)$$

Under the first analysis in which only the set of all observable prefixes were considered, the representation of the process would be the following set:

$$[\![P]\!]_\emptyset = \{\overline{in_S}\langle k_j\rangle, out_S(x_j), in_S(y), \overline{out_S}\langle D_1\rangle, \ldots, \overline{out_S}\langle D_N\rangle\}$$

and the analysis function $\Phi_{[\![P]\!]_\emptyset}$ iterated to a fixed point from bottom would compute the following solution:

$$\begin{array}{rclrcl} R(x_j) & = & \{D_1, \ldots, D_N\} & K(out_S) & = & \{D_1, \ldots, D_N\} \\ R(y) & = & \{k_j\} & K(in_S) & = & \{k_j\} \\ K(k_i) & = & \{\}\ \forall i & K(D_i) & = & \{\}\ \forall i \end{array}$$

and notice that this analysis now computes that all of the private data records for every user could potentially be sent to the client $C_j$ (because they were received into

$x_j$). If this analysis were used as the basis for determining the security of this trivial database design in terms of data ownership (i.e., if it were used to check if only the owner of a piece of data can obtain it), then the check would fail and this simple system would be deemed insecure with respect to this property. This is clearly not the case, as can be seen by the inspection of the traces of $P$ which are quite simple in this case: the process can only send the record $D_j$ to the client $C_j$ in any trace. This example serves to illustrate the drawbacks of the simple flow insensitive version of the analysis: it may reject secure processes.

Now, consider what happens to the same process when the match prefix is considered:

$$\begin{aligned}
[\![P]\!]_{\mathcal{M}} \;=\; \{&(\overline{in_S}\langle k_j\rangle, \emptyset),\\
&(out_S(x_j), \emptyset),\\
&(in_S(y), \emptyset),\\
&(\overline{out_S}\langle D_1\rangle, \{[y = k_1]\}),\\
&\qquad\vdots\\
&(\overline{out_S}\langle D_N\rangle, \{[y = k_N]\})\}
\end{aligned}$$

Notice that each prefix in the branches of the summation is paired with the unique match prefix that precedes. Thus the analysis should be able to exploit this to conclude that only one of them can possibly pass, and indeed $\mathrm{lfp}(\widetilde{\Phi}_{[\![P]\!]_{\mathcal{M}}})$ computes the following solution:

$$\begin{array}{rclrcl}
R(x_j) &=& \{D_j\} & K(out_S) &=& \{D_j\}\\
R(y) &=& \{k_j\} & K(in_S) &=& \{k_j\}\\
K(k_i) &=& \{\}\ \forall i & K(D_i) &=& \{\}\ \forall i
\end{array}$$

Note that the analysis now correctly computes that only the datum $D_j$ can be received by the client (i.e., $R(x_j) = \{D_j\}$).

## 7.3 Benefits of Sequential Analysis

This section continues the database example above to show the accuracy benefits of the sequential dataflow analysis presented in chapter 6. Consider what happens if the clients from the above process were modified to ping the server, and then only send their password if their initial communication were acknowledgement, i.e., the clients would formally be modified to be the following processes:

$$C'_i \triangleq \overline{in_S}\langle ping\rangle.ping(z_i).C_i$$

The client now sends an (arbitrary) name *ping* to the server's input channel, and waits to receive another (arbitrary) name *ping* now acting as a channel, before behaving as the previous client $C_i$. If a particular client $C'_j$ were now allowed to interact with the previous server in isolation:

$$P \triangleq (\nu\ ping)(\nu\ in_S)(\nu\ out_S)(\nu\ k_1, \ldots, k_N)(\nu\ D_1, \ldots, D_N)(S \| C'_j)$$

Then the representation of the process under the match-sensitive interpretation $[\![\cdot]\!]_{\mathcal{M}}$ would only add the two new prefixes to the set, i.e.,:

$$\begin{aligned}[\![P']\!]_{\mathcal{M}} &= \{(\overline{in_S}\langle ping\rangle, \emptyset), \\ &\qquad (ping(z_j), \emptyset)\} \cup [\![P]\!]_{\mathcal{M}}\end{aligned}$$

And the least solution of the dataflow analysis is given by the following sets:

$$\begin{array}{rclrcl} R(x_j) &=& \{D_j\} & K(out_S) &=& \{D_j\} \\ R(y) &=& \{k_j, ping\} & K(in_S) &=& \{k_j, ping\} \\ R(z_j) &=& \{\} & K(D_i) &=& \{\}\ \forall i \\ K(ping) &=& \{\} & K(k_i) &=& \{\}\ \forall i \end{array}$$

This analysis computes that the datum $D_j$ may reach the client, even though this is actually impossible (because the client never gets a return message on *ping*). In fact,

the analysis even shows that the client receives the datum *despite* not receiving the ping, because the $K$ set of the name *ping* is empty. This would violate the above property that the client should not send its password (which the analysis says it does because $k_j \in K(in_S)$) unless it receives an acknowledgement from the server first.

In order to catch this property, the fact that the second prefix in $C_i'$ cannot occur would need to be detected, and this can be done by using the fully sequential analysis from chapter 6. The representation of $P'$ under this analysis is as follows:

$$\begin{aligned}
[\![P']\!]_{\Pi} \;=\; & \{(\overline{in_S}\langle ping\rangle, \emptyset),\\
& (ping(z_j), \{\overline{in_S}\langle ping\rangle\}),\\
& (\overline{in_S}\langle k_j\rangle, \{\overline{in_S}\langle ping\rangle, ping(z_j)\}),\\
& (out_S(x_j), \{\overline{in_S}\langle ping\rangle, ping(z_j), \overline{in_S}\langle k_j\rangle\}),\\
& (in_S(y), \emptyset),\\
& ([y = k_1], \{in_S(y)\}),\\
& \qquad\qquad \vdots\\
& ([y = k_N], \{in_S(y)\})\\
& (\overline{out_S}\langle D_1\rangle, \{in_S(y), [y = k_1]\}),\\
& \qquad\qquad \vdots\\
& (\overline{out_S}\langle D_N\rangle, \{in_S(y), [y = k_N]\})\}
\end{aligned}$$

The top four entries represent the prefixes in the client (note how each one contains the predecessor set of the prefix before it). Now the sequential analysis is able to tell in the first iteration that the only two prefixes that can potentially communicate are $\overline{in_S}\langle ping\rangle$ and $\overline{in_S}\langle y\rangle$, because both of their predecessor sets are empty, and thus $\mathcal{C}[\rho[\pi]]$ will be set to true for both of them, and they are communication enabled. This iteration will cause the $\mathcal{C}$ function to be set to true for these prefixes, and in the next iteration, those prefixes that only contain one of these two in their predecessor sets

will be able to communicate if they are communication enabled. The only prefixes whose entire predecessor sets are mapped to $\Bbbk$ by $\mathcal{C}$ at this point are the $ping(z_j)$ prefix in the client, and the match prefixes $[y = k_i]$ in the server. Since $R(y)$ was set to $\{ping\}$ during the communication of the first iteration, and $R(k_i) = \{k_i\}$ for each $k_i$ by the initialization of $R$, none of th match prefixes are match-enabled. The input prefix $ping(z_j)$ has no other prefixes to communicate with whose predecessors have passed, thus the analysis adds no new information implying that the following fixed point is reached:

$$\begin{array}{rclrcl} R(x_j) & = & \{\} & K(out_S) & = & \{\} \\ R(y) & = & \{ping\} & K(in_S) & = & \{ping\} \\ R(z_j) & = & \{\} & K(D_i) & = & \{\}\ \forall i \\ K(ping) & = & \{\} & K(k_i) & = & \{\}\ \forall i \end{array}$$

Which reflects the more accurate result that the client's password is never transmitted, and the interaction of the client with the server is safe with respect to the above property. In fact, the solution accurately reflects that the only transmission that occurred was the ping.

# Chapter 8

# Future Work and Conclusions

This thesis concludes by pointing out some limitations of the analysis developed in the previous chapters, and discusses lines of research whereby some of these limitations could be addressed. Some of the work discussed in the following involves integration of previous work into the framework presented here and some involves new ideas that could potentially improve the analyses in various ways.

Revisiting the database query example from the previous chapter will serve to illustrate some of the limitations of the analysis as it stands:

$$\begin{aligned} C_i &\triangleq \overline{in_S}\langle k_i\rangle.out_S(x_i) \\ S &\triangleq in_S(y).\left(\sum_{i\in\{1,\ldots,N\}} [y = k_i].\overline{out_S}\langle D_i\rangle\right) \end{aligned}$$

In the previous chapter, only the behaviour of a single client asking for access from the server was analyzed. Consider now what occurs if all the clients are allowed to interact with the server at once:

$$P \triangleq (\nu\ in_S)(\nu\ out_S)(\nu\ k_1,\ldots,k_N)(\nu\ D_1,\ldots,D_N)(S\|C_1\|\cdots\|C_N)$$

Due to the non-determinism inherent in the $\pi$-calculus, even the most accurate analysis presented in this thesis produces undesirable results. Namely, the analysis from

section 6.2 where the parse tree of the process is analyzed to only consider communication between structurally compatible pairs produces the following least solution:

$$\begin{array}{rclrcl} R(x_1) & = & \{D_1, \ldots, D_N\} & K(out_S) & = & \{D_1, \ldots, D_N\} \\ & \vdots & & K(in_S) & = & \{k_1, \ldots, k_N\} \\ R(x_N) & = & \{D_1, \ldots, D_N\} & K(D_i) & = & \{\}\ \forall i \\ R(y) & = & \{k_1, \ldots, k_n\} & K(k_i) & = & \{\}\ \forall i \end{array}$$

This indicates that any of the clients could receive any of the private data (because $R(x_i) = \{D_1, \ldots, D_N\}$), even though any trace of the process only allows one of the clients to communicate with the server. This occurs because the server has a single channel for all inputs, and one for all outputs, thus the analysis conservatively computes that each of the keys could arrive on the input channel, and thus any of the data records could potentially be transmitted on the output channel. However, this result is inaccurate because no one trace of the process allows this to happen, it is only the amalgamation of the behaviours of all of the traces that lead to this result.

These (and other) drawbacks can be addressed in many ways, some of which are discussed in the following sections.

## 8.1 The Spi-Calculus: A Cryptographic Extension

The spi-calculus [AG99] is an extension of the $\pi$-calculus where, in addition to the set of names $\mathcal{N}$, encrypted messages of the form $\{M\}_K$ (representing the name $M$ encrypted with the key $K$) are included. Defining the set $\mathcal{H}$ as the set of encrypted names, encryption can be viewed as a function $\{\cdot\}_{[\cdot]} : \mathcal{N} \rightarrow \mathcal{N} \rightarrow \mathcal{H}$, which when given a name $K \in \mathcal{N}$ defines a family $\{\cdot\}_K : \mathcal{N} \rightarrow \mathcal{H}$ of functions that encrypt the input name with key $K$. This requires the addition of some construct used to decrypt names, and thus a new prefix is added to our syntax that allows us to do so:

$$\pi ::= \ldots \mid \mathtt{dec}_N(L \rightharpoonup x)$$

The new prefix tries to decrypt the name $L$ with key $N$. If $L$ is a name of the form $\{M\}_N$, then the process behaves as $P\{x \mapsto M\}$, otherwise it is stuck. Note that this is a slight variation in notation from Abadi and Gordon's decryption primitive in [AG99] and [AG98], but by definition $\mathtt{dec}_N(L \rightharpoonup x).P \equiv \mathit{case}\ L\ \mathit{of}\ \{x\}_N\ \mathit{in}\ P$ from the original description. It is more convenient for the analyses presented in this thesis to describe it as a prefix. It is important not to confuse the syntactic harpoon symbol "$\rightharpoonup$" with the semantic reduction relation "$\xrightarrow{\mu}$". The "$\rightharpoonup$" symbol in the decryption prefix is only intended to indicate that the result of a successful decryption of $L$ is substituted for the bound variable $x$ in $P$.

The semantics of the calculus only allow names in $\mathcal{H}$ to be used as transmitted data, and never as channels or data in an input prefix that could be substituted for. Furthermore the names in $\mathcal{H}$ are assumed to have the following properties [AG98]:

- The only way to decrypt an encrypted packet in $\{M\}_K \in \mathcal{H}$ is to know the corresponding key $K$.
- An encrypted packet does not reveal the key that was used to encrypt it.
- There is sufficient redundancy in messages (i.e., names in $\mathcal{N}$) to detect whether the decryption was successful

These assumptions imply that the cryptographic primitives used in the spi-calculus represent a perfect symmetric cryptosystem. In order to add support for the spi-calculus to the dataflow analyses presented in this thesis, it suffices to create a rule whereby a decryption prefix can successfully occur. This is analogous to the enabling rules for the other kinds of prefixes defined in chapter 6. Intuitively, a decryption enabling rule for the decryption prefix $\mathtt{dec}_N(L \rightharpoonup x)$ could conservatively go as follows:

- for every name of the form $\{M\}_L \in R(L)$ (i.e., the names that $L$ could be), the set $R(x)$ must include $M$.

It should be fairly trivial to add such a rule to the sequential analysis. If it were done, then the database example from above could be rewritten to return encrypted data to the clients:

$$\begin{aligned} C_i &\triangleq \overline{in_S}\langle k_i\rangle.out_S(x_i).\mathtt{dec}_{K_i}(x_i \rightharpoonup z_i) \\ S &\triangleq in_S(y).\left(\sum_{i\in\{1,\ldots,N\}} [y = k_i].\overline{out_S}\langle\{D_i\}_{K_i}\rangle\right) \end{aligned}$$

where the upper case $K_i$ represent an encryption key used by each client on its data (as opposed to the lower case $k_i$ representing an access password). The result of such an analysis would not change from the one shown at the start of the chapter, except that the set of names substituted for each $x_i$ (i.e., $R(x_i)$) would contain the encrypted version of the data packet, i.e., for each $i$:

$$R(x_i) = \{\{D_1\}_{K_1}, \ldots, \{D_N\}_{K_N}\}$$

and the subsequent decryption prefix now included in each client would only be able to decrypt the appropriate packet, thus the analysis would compute that

$$R(z_i) = \{D_i\}$$

for any $i$ thereby, showing that each client can only access its own data record. A fully flow insensitive analysis of the spi-calculus has already been developed in [BDNN01a], but it would be useful to integrate the sequential granularity and sensitivity to flow operators presented in the dataflow analyses here.

## 8.2 Refined Solution Space

The solution space $\mathcal{L}_\Psi$ of functions of the form $(R, K, E, \mathcal{C})$ can be refined to provide more granular information about the process. For instance, consider the flow-logic based analysis of Bodei *et al.* presented in appendix A.2. Rather than computing a function $K$ which only provides information about the names that were transmitted over a channel, the analysis computes a function $\widetilde{\kappa}$ that also takes as input a "sub-process" address (which represents a program point by the set of prefixes that precede a branch by a $\|$ or $+$ operator). The analysis is thereby able to provide information about what names were transmitted over which channels, *and at what program point the transmission may have occurred.*

Thanks to the precision of the sequential analysis in chapter 6, it is possible (and perhaps, even quite easy) to compute a solution space that is even more precise than this. Specifically, because the sequencing operator "." is fully considered, in that the analysis will not consider the contribution of prefixes that are blocked, it should be simple to compute a function $\widetilde{K} : \mathcal{O} \times \mathcal{N} \to \mathcal{N}$ that computes information about what names could be transmitted over a channel *by a particular prefix.*

This sort of granularity is already provided for the $R$ function, because a name $x$ can only appear as the datum in an input prefix once as per assumption 1.

Such information could then be used to reason about further security properties such as "authenticity", which deals with validating the identity of the processes that transmit to one another. This would be due to the fact that the analysis solution would now contain information about which prefix occurences transmitted what data, thus by determining the process containing each prefix, the identity of the transmitting process can be established.

## 8.3 Eliminating Approximation Points

The analyses presented in this thesis present a mechanism whereby the actual execution of the process is analyzed in a more accurate way than in the past; however, there are still conservative assumptions made by the techniques that could be improved without losing the decidability of the analysis. The first of these is the conservative nature of the conditions used to determine if a prefix is blocked, and the second is the analyses' treatment of replicated processes. The following subsections discuss the work to be done for each of these.

### 8.3.1 Blocking Conditions

Consider a prefix $\pi$ such that $\mathbf{pset}[\pi]$ contains a set of match prefixes $[x_1 = y_2], \ldots, [x_n = y_n]$. The dataflow analysis presented in the previous chapters determines that $\pi$ could fire if each of these matches passes under the condition

$$R(x_i) \cap R(y_i) \neq \emptyset$$

for each match $[x_i = y_i]$. This condition considers each match prefix individually, but it is quite possible for sequences of matches to be dependent on one another. For instance, consider if $\{[x = y], [y = z]\} \subseteq \mathbf{pset}[\pi]$, the current analysis would require that the following condition hold in order for these two matches to be enabled:

$$(R(x) \cap R(y) \neq \emptyset) \wedge (R(y) \cap R(z) \neq \emptyset) \tag{8.1}$$

However, this condition ignores the fact that in order for *both* matches to pass at once (which is indeed what is required), the following more restrictive condition suffices:

$$R(x) \cap R(y) \cap R(z) \neq \emptyset \tag{8.2}$$

Namely that if both matches are to pass, the set of names that $y$ could be must overlap the names that $z$ and $x$ could be at once. Notice that 8.2 implies 8.1, but the

reverse is not true in general, thus the first condition may determine that a prefix is not blocked, when in fact it is.

A forthcoming paper by Colussi *et al.* [CFG04] deals with such a refined blocking condition, but does not consider the environment of the process in doing so, and also provides no conditions under which output prefixes could block. Integrating that work into the framework presented here should improve the accuracy of the analyses.

### 8.3.2 Replication

The analysis as it stands may produce different results for structurally congruent processes. Consider a replicated sub-process $!P$ such that $bn(P)$ is not empty. Any $a \in bn(P)$ actually represents an infinite set of names $\{a_0, a_1 \ldots\}$. The current analysis identifies the potential solution sets of these names by combining them. This is done by representing a replicated sub-process as the set of its prefixes instead of reasoning about its sub-structure.

However, if the algorithm were given the syntactically distinct (but structurally congruent) process $P \| !P$ as input, the $\alpha$-conversion of the bound names would cause the names in the unfolded occurrence of $P$ to be re-named, thus potentially yielding non-trivial results for the new names and a slightly smaller solution for the set $\{a_0, a_1 \ldots\}$ (as the communications taken by the first unfolding of $P$ would no longer be included in the identification of the solutions for the replicated process).

This is not surprising, as a similar result applies to dataflow analyses working on inputs with rolled or unrolled loops in sequential programs. In order to explicitly relate these solutions (i.e., formally say that the solution for $P \| !P$ is less than the solution for $!P$ alone as desired), the analysis would need to keep track of the fact that $P$ "came from" $!P$ in some way. This would require a semantics that allows such things to be tracked, and some work on such non-standard semantics for the calculus

has been done by Feret [Fer01].

Furthermore, while the current analysis has demonstrated useful techniques for dealing with sequences of prefixes, it would be desirable to apply similar techniques to obtain more accurate results for replicated processes. This could be done if the unfolding processes in a replicated process were analyzed individually with the same techniques. However, in order to keep the analysis from becoming undecidable, it may be necessary to "cap" the number of times that a particular replicated process is allowed to unroll. Such an analysis may also require additional constructs to be added to the calculus to indicate when the unfoldings occur. For example, instead of modelling the unfolding of a replicated process in the structural congruence relation, one could possibly add a "signalling" construct to the syntax that would cause a particular replicated subprocess to unfold a copy during reduction. These ideas are, however, totally preliminary, thus the usefulness or feasibility of such an extension to the language is not known.

## 8.4 Full Context Independence

The analysis presented provides a partial context independence result for the $R$ component of the solution. The result is partial because it only considers the possibility of the analyzed process in parallel composition with another process, i.e., contexts of the form $C \parallel [\cdot]$. This could be potentially expanded in two ways:

1. Generating a $K$ component that is also independent of concurrent processes
2. Generating results that are also independent of arbitrary contexts

The first point was briefly discussed in section 5.1.2 (the proof of the correctness of the $\Phi$ function). That proof illustrated the fact that the only reason the $K$ function is not independent of a concurrent context is because of the possibility of the context

transmitting names over channels that were in the free names of the analyzed process, or names that were extruded by the process. This *could* be fixed in the following way:

- setting the $K$ set of every name in $\text{fn}(P)$ to $\bot_E$ initially
- setting the $K$ set of each name in $E$ that is also in $P$ to the current environment knowledge $E$ every time a name is extruded

While these improvements are possible, their usefulness is in question, as such information no longer provides information about the dataflow through the analyzed process $P$, and may potentially trivialize the generated results.

The applicability of generating results that are independent of arbitrary contexts is also questionable. For instance, an arbitrary context could bind names that are free in $P$, but the question then shifts to the interpretation of what such a context would represent in terms of applications such as security. Concurrent contexts such as $C \parallel [\cdot]$ model the ability of an adversary to potentially interact with a process by communicating with it, but what does a context like $(\nu b)\ [\cdot]$, or $x(y).[\cdot]$ model in terms of security? Such questions need to be answered before it is determined whether generating results independent of such contexts actually have any practical worth.

## 8.5 Efficiency

The basic analysis of chapter 4 was shown to be computable by an algorithm similar to an Andersen-style points-to analysis that runs in polynomial time. All of the other analyses in the thesis are computable with similar algorithms because they only restrict the prefixes that are iterated based on polynomially decidable conditions, and thus even the most accurate analysis presented here is still polynomial. However,

the question of how to compute these functions more *efficiently* has been completed avoided.

There has been much work in the literature on the question of efficiency. Bodei *et al.*'s match-sensitive analysis (i.e., the version of the function $\widetilde{\Phi}$ that doesn't consider the environment) has been shown to be computable in cubic time by Nielson and Seidl [NS01] by expressing the analysis as a set of Horn Clauses and subsequently using a standard algorithm to solve them. The forthcoming paper by Colussi *et al.* [CFG04] (mentioned above) also provides an algorithm to compute its analysis in cubic time. This is quite surprising because their analysis considers blocking matches and input prefixes, and is thus much more accurate than the one discussed by Nielson and Seidl.

Adapting these techniques to the analyses presented in this thesis should provide means of developing efficient algorithms for the accurate and context independent analyses developed here.

# Appendix A
# Static Analysis Using Flow Logics

Bodei *et al.* [BDNN98, BDNN01b, BDPZ03] approach the problem of statically analyzing process algebrae by using flow logics. A solution is characterized by presenting a judgement determining the correctness of a proposed solution with respect to a given process. The advantage of such a system is that correctness can effectively be proven by proving a fairly standard subject reduction result for the judgement with respect to the operational semantics.

## A.1 Flow Insensitive Analysis

The analysis in [BDNN98, BDNN01b] computes a pair of functions $(\rho, \kappa)$ for a given process $P$. Here $\rho : \mathcal{N} \to 2^{\mathcal{N}}$ maps a name bound by input in $P$ to a set of names that it could potentially assume during execution, and $\kappa : \mathcal{N} \to 2^{\mathcal{N}}$ maps a name not bound by an input prefix to the set of names that could potentially transmitted over it during the reduction of $P$.

Given a proposed solution $(\rho, \kappa)$, the validity of the solution is checked by deriving a judgement of the form

$$(\rho, \kappa) \models P$$

| | | |
|---|---|---|
| $(\rho,\kappa) \models \mathbf{0}$ | iff | $\mathtt{tt}$ |
| $(\rho,\kappa) \models \tau.P$ | iff | $(\rho,\kappa) \models P$ |
| $(\rho,\kappa) \models \overline{x}\langle y\rangle.P$ | iff | $(\rho,\kappa) \models P \wedge$ |
| | | $\forall u \in \rho(x) : \rho(y) \subseteq \kappa(u)$ |
| $(\rho,\kappa) \models x(y).P$ | iff | $(\rho,\kappa) \models P \wedge$ |
| | | $\forall u \in \rho(x) : \kappa(u) \subseteq \rho(y)$ |
| $(\rho,\kappa) \models [x=y].P$ | iff | $(R(x) \cap R(y)) \neq \emptyset \Rightarrow (\rho,\kappa) \models P$ |
| $(\rho,\kappa) \models P_1 + P_2$ | iff | $(\rho,\kappa) \models P_1 \ \wedge \ (\rho,\kappa) \models P_2$ |
| $(\rho,\kappa) \models P_1 \| P_2$ | iff | $(\rho,\kappa) \models P_1 \ \wedge \ (\rho,\kappa) \models P_2$ |
| $(\rho,\kappa) \models (\nu x)P$ | iff | $(\rho,\kappa) \models P$ |
| $(\rho,\kappa) \models !P$ | iff | $(\rho,\kappa) \models P$ |

Table A.1: Flow Insensitive CFA for the Π-Calculus (modified from [BDNN01b])

using the rules provided in table A.1. Note that these rules have been modified from [BDNN98] to fully utilize the disciplined $\alpha$-conversion assumption made in section 3.2.1. The modifications made only involve removing extreaneous labels from the terms, and do not affect any of the results published in [BDNN98, BDNN01b].

Correctness here informally means that if $(\rho,\kappa) \models P$ (i.e., $(\rho,\kappa)$ is a valid approximation of $P$'s behaviour), then any communication that could occur during any possible execution (i.e., reduction sequence) of $P$ is faithfully modelled by $(\rho,\kappa)$. In other words, if it is possible, in some execution sequence, that the name $y$ is sent on channel $x$ by some prefix $\overline{x}\langle y\rangle$, then $y \in \kappa(x)$. Formally, this result is obtained by proving a subject reduction theorem like the following (again modified from [BDNN98, BDNN01b]):

**Theorem A.1.1.** *If* $(\rho,\kappa) \models P$ *and* $P \xrightarrow{\mu} Q$ *then:*

| | | | | |
|---|---|---|---|---|
| **(1)** | *if* | $\mu = \tau$ | *then* | $(\rho,\kappa) \models Q$; |
| **(2a)** | *if* | $\mu = \overline{x}y$ | *then* | $(\rho,\kappa) \models Q \ \wedge \ (y \in \kappa(x))$; |
| **(2b)** | *if* | $\mu = \overline{x}(y)$ | *then* | $(\rho,\kappa) \models Q \ \wedge \ (y \in \kappa(x))$; |
| **(3)** | *if* | $\mu = xy$ | *then* | $(y \in \kappa(x)) \ \Rightarrow \ (\rho,\kappa) \models Q$. |

*Proof.* By induction on the construction of $P \xrightarrow{\mu} Q$. See theorem 3.10 in [BDNN01b] for details. □

In addition to the characterization, it was proved that the set of possible solutions $(\rho, \kappa)$ of the analysis constitutes a Moore family, implying that a least (i.e., most accurate) correct solution exists. Bodei *et al.* provided an algorithm whereby this solution could be constructed by iteratively checking a set of constraints, and the efficiency of this procedure was improved by Nielson and Seidl [NS01] by formulating the problem as a set of Horn clauses with sharing. Note that the above theorem does not consider the context that $P$ could be running in, this drawback is discussed and corrected in chapter 5.

## A.2 Flow Sensitive Analysis

The control flow analysis above only considered the prefixes contained in a process description when computing its approximation: constraints are only added to the characterization in table A.1 when an input or output prefix *not guarded by a blocked match prefix* is encountered. In fact, this approach does not distinguish between any process operators: consider the following processes:

$$\begin{aligned} P_1 &\triangleq \overline{a}\langle c\rangle \parallel a(b).\overline{b}\langle c\rangle + a(b) \\ P_2 &\triangleq \overline{a}\langle c\rangle \parallel a(b) \parallel \overline{b}\langle c\rangle \parallel a(b) \end{aligned}$$

The least solutions $(\rho_1, \kappa_1)$ and $(\rho_1, \kappa_2)$ such that $(\rho_1, \kappa_1) \models P_1$ and $(\rho_2, \kappa_2) \models P_2$ are in fact identical. The analysis takes neither the sequence (.), composition ($\parallel$), nor choice (+) operators into account. Consequently, given a process such as $x(y) + \overline{x}\langle z\rangle$, the analysis would compute that $z$ could be transmitted over $x$ (i.e., that $z \in \kappa(x)$) which is not a possible reduction for the process in question. Therefore, while

$\forall a, b \in \mathcal{N}$,
$\forall Y \subseteq \mathcal{N}$ such that $(|Y| \neq \infty) \vee (Y = \mathcal{N})$,

| for $\vartheta \neq \epsilon$: | | | $\forall \vartheta$: | | |
|---|---|---|---|---|---|
| $\overline{a}\langle b\rangle.P@\vartheta$ | $=$ | $P@\vartheta$ | $P@\epsilon$ | $=$ | $P$ |
| $a(b \in Y).P@\vartheta$ | $=$ | $P@\vartheta$ | $(P_0 \| P_1)@\|_i\vartheta$ | $=$ | $P_i@\vartheta$ |
| $((\nu a)P)@\vartheta$ | $=$ | $P@\vartheta$ | $(P_0 + P_1)@+_i\vartheta$ | $=$ | $P_i@\vartheta$ |
| $\tau.P@\vartheta$ | $=$ | $P@\vartheta$ | | | |

Table A.2: Localization Operator on Processes

ignoring process operators leads to a fast algorithm for computing the least solution, it is also very conservative: a more accurate solution may be desired. Accuracy can be improved by observing that prefixes can only communicate if they are on opposite sides of a compsition operator, or on the same side of a choice operator. This information is tracked in [BDPZ03] by defining an operator $@\vartheta$ recursively on the structure of processes as in table A.2. Note that the results in that paper were discussed in the context of a modified calculus which omits the match prefix in favor of a "selective input" prefix $x(y \in Y)$. The semantics of the latter are identical to that of the usual input prefix that only synchronizes if the received name is in the set $Y$.

Given a process $P$, $P@\vartheta$ refers to the subprocess of $P$ that has address $\vartheta$. The symbol $\epsilon$ is defined as the empty address, and thus $P@\epsilon$ is $P$ for any process $P$. Addresses $\vartheta$ are finite strings over the alphabet $\{+_i, \|_i\}_{i\in\{0,1\}}$. The set $Addr(P) = \{\vartheta \mid \exists Q : P@\vartheta = Q\}$ is defined as the set of all sub-process addresses of the process $P$. This definition assigns an address to every program point. The set $\mathcal{A}$ represents the set of all possible addresses, i.e., $\mathcal{A} = \{+_0, +_1, \|_0, \|_1\}^*$.

In order to track which subprocesses are able to communicate (i.e., those on opposite sides of a $\|$ or on the same side of a $+$), the notion of *compatibiliity* of subprocesses is defined formally in [BDPZ03] as follows:

**Definition A.2.1.** Given a process $P$ and two addresses $\vartheta, \vartheta' \in Addr(P)$, $\vartheta$ and $\vartheta'$ are *compatible*, written as $\mathtt{comp}_P(\vartheta, \vartheta')$, if and only if

$$\begin{aligned} \vartheta &= \vartheta_0 \|_i \vartheta_1, \text{ and} \\ \vartheta' &= \vartheta_0 \|_{1-i} \vartheta_1' \end{aligned}$$

for $i \in \{0, 1\}$ and some $\vartheta_1, \vartheta_1'$.

This scheme encodes the ability of various program points in a $\pi$-calculus process to communicate. This sensitivity to the choice and composition operators will allow the accuracy of the analysis shown in table A.1 to be improved. The improved control flow analysis computes a pair of functions $(\widetilde{\rho}, \widetilde{\kappa})$ – where $\widetilde{\rho}$ is as $\rho$ from above, and $\widetilde{\kappa} : (\mathcal{A} \times \mathcal{N}) \to 2^{\mathcal{N}}$ is a more granular version of $\kappa$ which associates a name *at a particular subprocess address* with a set of names which could be transmitted over it. Note that these functions are again generalized to sets in the obvious way. The flow logic characterization of the analysis solution is presented in table A.3. Observe that this analysis carries the address of the current suprocess with it at each step. An estimate for a process $P$ is the pair $(\widetilde{\rho}, \widetilde{\kappa})$ such that $(\widetilde{\rho}, \widetilde{\kappa}) \models^{\epsilon} P$, and $\mathtt{comp} = \mathtt{comp}_P$.

Note that this characterization differs from the flow insensitive version in table A.1 only in that the condition the constraints from input prefixes are only imposed for subprocesses which are compatible with the current subprocess. Given the least solutions $(\widetilde{\rho}, \widetilde{\kappa})$ such that $(\widetilde{\rho}, \widetilde{\kappa}) \models^{\epsilon} P$ and $(\rho, \kappa)$ such that $(\rho, \kappa) \models P$ for some process $P$, then it is easy to prove that $(\widetilde{\rho}, \bigsqcup_{\mathcal{A}} \widetilde{\kappa}(\vartheta, -)) \sqsubseteq (\rho, \kappa)$ (with $\bigsqcup$ and $\sqsubseteq$ the standard pointwise supremum and orderings on functions), and that in fact this inequality is strict for appropriately chosen $P$ (such as the process $x(y) + \overline{x}\langle z \rangle$ discussed above). Thus this analysis is strictly more accurate than the previous flow insensitive CFA (see [BDPZ03] for further information).

The contextual dependency alluded to in the previous section is partially handled in [BDPZ03] by also tracking the potential environment knowledge of the reducing

| | | |
|---|---|---|
| $(\widetilde{\rho},\widetilde{\kappa}) \models^{\vartheta} \mathbf{0}$ | iff | $\mathbb{t}$ |
| $(\widetilde{\rho},\widetilde{\kappa}) \models^{\vartheta} \tau.P$ | iff | $(\widetilde{\rho},\widetilde{\kappa}) \models^{\vartheta} P$ |
| $(\widetilde{\rho},\widetilde{\kappa}) \models^{\vartheta} \overline{x}\langle y\rangle.P$ | iff | $(\widetilde{\rho},\widetilde{\kappa}) \models^{\vartheta} P \wedge$ |
| | | $\forall u \in \widetilde{\rho}(x) : \widetilde{\rho}(y) \subseteq \widetilde{\kappa}(\vartheta, u)$ |
| $(\widetilde{\rho},\widetilde{\kappa}) \models^{\vartheta} x(y \in Y).P$ | iff | $(\widetilde{\rho},\widetilde{\kappa}) \models^{\vartheta} P \wedge$ |
| | | $\forall u \in \widetilde{\rho}(x) : \forall \vartheta' : \mathtt{comp}(\vartheta, \vartheta') :$ |
| | | $(\widetilde{\kappa}(\vartheta', u) \cap \widetilde{\rho}(Y)) \neq \emptyset \Rightarrow$ |
| | | $(\widetilde{\kappa}(\vartheta', u) \cap \widetilde{\rho}(Y)) \subseteq \widetilde{\rho}(y)$ |
| $(\widetilde{\rho},\widetilde{\kappa}) \models^{\vartheta} P_1 + P_2$ | iff | $(\widetilde{\rho},\widetilde{\kappa}) \models^{\vartheta +\!\!+_0} P_0 \ \wedge \ (\widetilde{\rho},\widetilde{\kappa}) \models^{\vartheta +\!\!+_1} P_1$ |
| $(\widetilde{\rho},\widetilde{\kappa}) \models^{\vartheta} P_1 \| P_2$ | iff | $(\widetilde{\rho},\widetilde{\kappa}) \models^{\vartheta \|_0} P_0 \ \wedge \ (\widetilde{\rho},\widetilde{\kappa}) \models^{\vartheta \|_1} P_1$ |
| $(\widetilde{\rho},\widetilde{\kappa}) \models^{\vartheta} (\nu x)P$ | iff | $(\widetilde{\rho},\widetilde{\kappa}) \models^{\vartheta} P$ |

Table A.3: Flow Sensitive CFA for the Π-Calculus (modified from [BDPZ03])

process similarly to the analysis of chapter 5. However, [BDPZ03] provides neither proof of correctness, nor constructive algorithms, although their existence is conjectured.

# Bibliography

[ABN89] M. Abadi, M. Burrows, and R.M. Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989.

[AFF97] A. Aiken, M. Fähndrich, 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.

[AFF99] Alexander Aiken, Manuel Fähndrich, and Jeffrey S. Foster. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*, Atlanta, Georgia, Mar 1999.

[AFF00] Alexander Aiken, Manuel Fähndrich, and Jeffrey S. Foster. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Static Analysis Symposium 2000*, Santa Barbara, California, Jun 2000.

[AFFC98] Alexander Aiken, Manuel Fähndrich, Jeffrey S. Foster, and Jason Cu. Tracking down exceptions in Standard ML programs. Technical Report 98-96, University of California, Berkeley, Computer Science Division, Feb 1998.

[AFFS98a] A. Aiken, M. Fähndrich, J.S. Foster, and Z. Su. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[AFFS98b] Alexander Aiken, Manuel Fähndrich, Jeffrey S. Foster, and Zhendong Su. A toolkit for constructing type- and constraint-based program analyses. In X.Leroy and A. Ohori, editors, *Proceedings of the second International Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 78–96. Springer-Verlag, Kyoto, Japan, Mar 1998.

[AFS00] Alexander Aiken, Manuel Fähndrich, and Zhendong Su. Detecting races in relay ladder logic programs. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2000.

[AG97] M. Abadi and A. D. Gordon. Reasoning about cryptographic protocols in the spi-calculus. In *CONCUR '97: Concurrency Theory*, volume 1243 of *Lecture Notes In Computer Science*, pages 59–73. Springer-Verlag, 1997.

[AG98] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols - the spi-calculus. Systems Research Center Report 149, Digital Equipment Corporation, January 1998.

[AG99] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols - the spi-calculus. *Information and Computation*, 148:1–70, January 1999.

[Aga00] J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, Jan 2000.

[AM98] S. Abramsky and G. McCusker. Game semantics. In U.Berger and H.Schwichtenberg, editors, *Computational Logic*, volume 165 of *NATO Science Series, Series F: Computer and Systems Sciences*, pages 1–55. Springer-Verlag, Berlin, Germany, 1998.

[And99] Lars Ole Andersen. *Program Analysis And Specialization for the C Programing Language*. PhD thesis, University Of Copenhagen, May 1999.

[AW92] Alexander Aiken and E.L. Wimmers. Solving systems of set constraints. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 329–340, Santa Cruz, California, Jun 1992.

[AW93] Alexander Aiken and E.L. Wimmers. Type inclusion constraints and type inference. In *FPCA '93 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, Jun 1993.

[Bar84] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[BB92] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, April 1992.

[BDER97] P. Baillot, V. Danos, T. Ehrhard, and L. Regnier. AJM's games model is a model of classical linear logic. In *Proceedings of the 12th Annual IEEE Symposium on Logic In Computer Science*, pages 68–75, Warsaw, Jun 1997. IEEE Computer Society Press.

[BDNN98] C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Control flow analysis for the $\pi$-calculus. In *Proceedings of CONCUR '98*, volume 1466

of *Lecture Notes In Computer Science*, pages 84–98. Springer-Verlag, 1998.

[BDNN01a] C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Static analysis for secrecy and non-interference in networks of processes. In *Proceedings of the* $6^{th}$ *International Conference on Parallel Computing Technologies*, pages 27–41, September 2001.

[BDNN01b] C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Static analysis for the $\pi$-calculus with applications to security. *INFCTRL: Information and Computation (formerly Information and Control*, 168, 2001.

[BDPZ03] C. Bodei, P. Degano, C. Priami, and N. Zannone. An enhanced CFA for security policies. In *Proceedings of the Workshop on Issues on the Theory of Security (WITS '03)*, pages 131–145, Warszawa, 2003.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points. In *ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CFG04] L. Colussi, G. Filè, and A. Griggio. Precise analysis of $\pi$-calculus in cubic time, *to be published*. In *Proceedings of the* $3^{rd}$ *IFIP International Conference on Theoretical Computer Science*, August 2004.

[CG89] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

[CGG02] Luca Cardelli, Giorgio Ghelli, and Andy Gordon. Secrecy and group creation. In *Electronic Notes in Theoretical Computer Science*, volume 40. Elsevier, 2002.

[CX02] K. Cooper and L. Xu. An efficient static analysis algorithm to detect redundant memory operations. In *Proceedings of the Workshop on Memory System Performance*, pages 97–107, 2002.

[DDMM03] M. Debbabi, N. Durgin, M. Mejri, and J. Mitchell. Security by typing. *Software Tools for Technology Transfer (STTT)*, 4(4):472–495, 2003.

[Fer00] Jérôme Feret. Confidentiality analysis of mobile systems. In *Proceedings of the $7^{th}$ International Static Analysis Symposium (SAS '00)*, volume 1824 of *Lecture Notes In Computer Science*. Springer-Verlag, 2000.

[Fer01] Jérôme Feret. Occurence counting analysis for the $\pi$-calculus. In *Electronic Notes in Theoretical Computer Science*, volume 39. Elsevier Science Publishers, 2001.

[FHMV95] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HR98] N. Heintze and J.G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 365–377, Jan 1998.

[HS02] D. Hughes and V. Shmatikov. Information hiding, anonymity and privacy: A modular approach. *Journal of Computer Security*, 2002. URL: <http://boole.stanford.edu/~dominic/papers/kripke.html>.

[MH98a] P. Malacaria and C. Hankin. Generalised flowcharts and games. In *Proceedings of the 25th International Colloquim on Automata, Languages, and Programming*, 1998.

[MH98b] P. Malacaria and C. Hankin. A new approach to control flow analysis. In *Computational Complexity*, pages 95–108, 1998.

[MH99] 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.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mit96] John Mitchell. *Foundations of Programming Languages*. MIT Press, 1996.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, 1992.

[NHN03] Flemming Nielson, René Rydhof Hansen, and Hanne Riis Nielson. Abstract interpretation of mobile ambients. *Science of Computer Programming*, 47:145–175, 2003.

[NNH99] F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[NNHJ99] 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.

[NS01] F. Nielson and H. Seidl. Control-flow analysis in cubic time. In *Proc. ESOP '01*, number 2028 in Lecture Notes in Computer Science, pages 252–268. Springer-Verlag, 2001.

[OPS92] Nicholas Oxhöj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. In Ole Lehrmann Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 329–349. Springer-Verlag, New York, N.Y., 1992.

[Par81] David Park. Concurrency and automata on infinite sequences. In *Proceedings of the* $5^{th}$ *GI-Conference on Theoretical Computer Science*, pages 167–183. Springer-Verlag, 1981.

[PR94] Hemant D. Pande and Barbara G. Ryder. Static type determination for C++. In *C++ Conference*, pages 85–98, 1994.

[PR96] H. D. Pande and B. G. Ryder. Data-flow-based virtual function resolution. *Lecture Notes in Computer Science*, 1145:238–50, 1996.

[PS95] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1995.

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices*, 35(10):264–280, 2000.

[SM03] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[SS99] P. Syverson and S. Stubblebine. Group principals and the formalization of anonymity. In *Proceedings of the World Congress on Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 814–833. Springer-Verlag, 1999.

[Sun99] Vijay Sundaresan. Practical techniques for virtual call resolution in Java. Master's thesis, McGill University, School of Computer Science, June 1999.
URL: <`http://www.sable.mcgill.ca/publications/`>.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3), 1996.

[Zim00] Pascal Zimmer. On the expressiveness of pure mobile ambients. In $7^{th}$ *International Workshop on Expressiveness in Concurrency*, volume 39 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.