Behavioral Equivalence in the Polymorphic Pi-Calculus

Benjamin C. Pierce
Computer Science Department
Indiana University
Lindley Hall 215
Bloomington, IN 47405, USA
pierce@cs.indiana.edu

Davide Sangiorgi
INRIA-Sophia Antipolis
2004 Rue des Lucioles
B.P. 93
06902 Sophia Antipolis, France
davide.sangiorgi@sophia.inria.fr

Indiana University Computer Science
Technical Report TR 468

October 2, 1996

Abstract

We investigate *parametric polymorphism* in message-based concurrent programming, focusing on behavioral equivalences in a typed process calculus analogous to the polymorphic lambda-calculus of Girard and Reynolds.

Polymorphism constrains the power of observers by preventing them from directly manipulating data values whose types are abstract, leading to notions of equivalence much coarser than the standard untyped ones. We study the nature of these constraints through simple examples of concurrent abstract data types and develop basic theoretical machinery for establishing bisimilarity of polymorphic processes.

We also observe some surprising interactions between polymorphism and aliasing, drawing examples from both the polymorphic pi-calculus and ML.

1 Introduction

We study the effect of adding polymorphic typing (in the style of the polymorphic lambda-calculus [Gir72, Rey74]) to the pi-calculus, a pure calculus of message-based concurrency [MPW92, Mil91]. This extension is syntactically quite straightforward — especially since a variety of type systems based on existing typed lambda-calculi have already been given for the pi-calculus — and standard metatheoretic properties such as subject reduction are easily proved [Tur96]. However, the effect of polymorphism on behavioral properties of programs poses more challenging problems, and here standard tools from the lambda-calculus are less useful; in particular, "the value of an expression" loses its simple sense in a world of communicating agents. To our knowledge, the present study is the first to consider the behavioral consequences of polymorphism in a concurrent setting.

Our primary interest is in the semantic concept of *parametric polymorphism*, a term coined by Strachey [Str67] for polymorphic functions that behave uniformly in their type arguments. In the polymorphic lambda-calculus, *all* polymorphic functions are parametric, since the calculus contains no operations for testing the actual types passed as parameters. Similarly, in the polymorphic pi-calculus, when a value communicated along a channel has a (partially or completely) abstract type, the usage of the value by a well-typed receiver must be independent from the hidden part of

the type, about which no assumptions can be made. We develop proof techniques for behavioral properties of polymorphic processes based on this intuition. (We have not established a formal connection between our techniques and Reynolds's notion of relational parametricity [Rey83], but the intent is similar.)

The Pi-Calculus

To define the polymorphic pi-calculus, we begin from the simply typed pi-calculus, an explicitly typed variant of Milner's polyadic pi-calculus with simple sorts [Mil91]. We briefly review this system and then show how polymorphic types are added.

The conceptual world of the pi-calculus comprises two sorts of entity: *processes*, which compute in parallel and exchange information by communication, and the *channels* on which they communicate. The power of the calculus arises in part from the fact that channels are not only a communication medium but may also be communicated as data along other channels, thus allowing dynamic reconfiguration of communication topologies. In particular, the ability to send freshly created channels to other processes can be used to model the passing of continuations in conventional functional programming. The most important combinators for building processes are these (the full syntax is defined formally in Section 2):

P	::=	$\overline{a}[b_1 \ldots b_n].\, P$	send $b_1 \ldots b_n$ along a and then become P
		$a(x_1 \ldots x_n).\, P$	receive $x_1 \ldots x_n$ along a and then become P
		$P \mid Q$	run P and Q in parallel
		$(\nu\, x{:}T\,)P$	create a fresh channel and call it x in P
		0	do nothing.

For example, suppose b is a channel. The process

$$\mathit{True} \stackrel{\text{def}}{=} b(t, f).\, \overline{t}[\,].\, 0$$

inputs a pair of channels, t and f, along b and then sends an empty tuple of channels along t. Similarly,

$$\mathit{False} \stackrel{\text{def}}{=} b(t, f).\, \overline{f}[\,].\, 0$$

reads t and f from b and then signals along f. The type of b in both of these processes is $b \in \updownarrow[\updownarrow[\,], \updownarrow[\,]]$, read "$b$ is a channel carrying pairs of channels, each of which carries empty tuples." (Since we use the type $\updownarrow[\updownarrow[\,], \updownarrow[\,]]$ often in what follows, we introduce the abbreviation *Bool* for it.) Restricting our attention to channel types of this simple form, where a given channel always carries tuples of the same shape, means that the typing rules for processes are completely straightforward and static — very similar, in fact, to the simply typed lambda-calculus.

True and *False* can be thought of as representing the two boolean values, in the sense that we can write a testing process that chooses between two alternative behaviors depending on whether it is placed in parallel with *True* or with *False*:

$$\begin{aligned}\mathit{Test} \stackrel{\text{def}}{=}\ & (\nu\, x{:}\updownarrow[\,])(\nu\, y{:}\updownarrow[\,]) \\ & (\overline{b}[x, y].\, 0 \\ & \mid x().\, R_1 \\ & \mid y().\, R_2)\end{aligned}$$

The first action of *Test* is to create two fresh channels x and y of appropriate type. It sends these along b and, in parallel, waits for inputs along x and y. If *Test* is placed in parallel with *True*, the

composite process performs the following sequence of interactions: first *Test* creates x and y and passes them to *True* along b; then b responds by sending an empty tuple along x, which is received by the second subprocess in the parallel composition in *Test*, after which R_1 is permitted to run. Since no signal will ever be sent along y (which was created fresh at the beginning and has never been told to anyone except *True*, which is never going to use it), the subprocess $y().\,R_2$ in *Test* is garbage.

Polymorphism

Another simple example, illustrating how a functional programming style can be imitated in the pi-calculus, is the process

$$Id \stackrel{\text{def}}{=} i(x, r).\,\overline{r}[x].\,0$$

which reads x and r from the channel i and then sends x along r. If we adopt the convention that a "client" of *Id* always sends a fresh channel r and then waits for *Id* to send its result back along r

$$IdClient \stackrel{\text{def}}{=} (\nu\, r{:}\updownarrow[\,Bool\,])(\overline{i}[x, r].\,r(y).\,\ldots)$$

then we may regard r as the continuation of the invocation of *Id*.

In order for *IdClient* to be well typed, the channel i must have type $\updownarrow[\,Bool, \updownarrow[\,Bool\,]\,]$. If we also want to use an identity function to manipulate a different type T, we must make up a different channel i' of type $\updownarrow[\,T, \updownarrow[\,T\,]\,]$ and use it to communicate with a different *Id* process, identical with the first one except for the channel it uses to receive arguments. This proliferation of nearly identical copies of *Id* cries out for a polymorphic extension of the type system.

In the interest of harmonious design, the facilities for type abstraction and instantiation that we introduce should follow the spirit of the pi-calculus, where all exchange of information is by communication between processes on channels. The polymorphic pi-calculus achieves this by making each communication include not only a tuple of values but also a tuple of types. For example, the polymorphic identity function is implemented by the process

$$PolyId \stackrel{\text{def}}{=} i\{X\}(x, r).\,\overline{r}[x].\,0$$

where the type of i is now written $\updownarrow\{X\}[X, \updownarrow[\,X\,]]$ — that is, each communication on i consists of a type X, a value of type X, and a continuation channel carrying values of type X. In the body of *PolyId*, x has type X and r has type $\updownarrow[\,X\,]$, so the output $\overline{r}[x]$ is consistently typed. A client of *PolyId* must now send a type in addition to the other two arguments:

$$PolyIdClient \stackrel{\text{def}}{=} (\nu\, r{:}\updownarrow[\,T\,])(\overline{i}\{T\}[x, r].\,r(y).\,\ldots)$$

In the terminology of the polymorphic lambda-calculus, we can think of each communication on the channel i as consisting of an existential package of type $\exists X.[X, \updownarrow[\,X\,]]$. Indeed, the typing rules in Section 3 for polymorphic output and input will correspond precisely to the usual existential introduction and elimination rules of the polymorphic lambda-calculus. (It is interesting to note that the natural primitive form of polymorphism in the pi-calculus is existential, not universal, quantification!)

Abstract Data Types

As in the lambda-calculus, polymorphism can be used in the pi-calculus to ensure that different parts of a program cannot directly manipulate the internal representations of each other's data structures. The following example illustrates the use of polymorphic typing to construct *abstract data types* in the pi-calculus, following Mitchell and Plotkin's encoding of ADTs in the polymorphic lambda-calculus [MP88].

Suppose we wish to implement boolean values as processes in the same way as above, but keeping hidden the details of the protocol used to implement boolean values and conditionals. We can achieve this by (1) providing conditional testing via an additional channel *test* of type $\updownarrow[\mathit{Bool}, \updownarrow[\,], \updownarrow[\,]]$, so that a client can test a boolean value b by sending it to *test*, along with two alternative continuation channels, instead of communicating directly with b; and (2) abstracting the types of t, f, and *test*, so that the *only* thing a client can do with a boolean is to pass it to *test*. This is done by making the channels t, f, and *test* private (local) and exporting them to clients by sending them along a channel *getBools* of polymorphic type $\updownarrow\{X\}[X, X, \updownarrow[X, \updownarrow[\,], \updownarrow[\,]]]$:

$$
\begin{array}{lll}
B_1 & \stackrel{\text{def}}{=} & (\nu\, t{:}\mathit{Bool},\ f{:}\mathit{Bool},\ \mathit{test}{:}\updownarrow[\mathit{Bool}, \updownarrow[\,], \updownarrow[\,]]\,) \\
& & \quad (\overline{\mathit{getBools}}\{\mathit{Bool}\}[t, f, \mathit{test}] \\
& & \quad \mid t(x, y).\, \overline{x}[].\, 0 \\
& & \quad \mid f(x, y).\, \overline{y}[].\, 0 \\
& & \quad \mid \mathit{test}(b, x, y).\, \overline{b}[x, y].\, 0)
\end{array}
$$

A client that wants to use the booleans must first obtain them from the "boolean server" B_1 by reading from *getBools*, and can then only communicate directly along *test*, since its type for t and f is just X; the actual type *Bool* has been hidden. An example of a different implementation of the boolean package is the process

$$
\begin{array}{lll}
B_2 & \stackrel{\text{def}}{=} & (\nu\, t{:}\mathit{Bool},\ f{:}\mathit{Bool},\ \mathit{test}{:}\updownarrow[\mathit{Bool}, \updownarrow[\,], \updownarrow[\,]]\,) \\
& & \quad (\overline{\mathit{getBools}}\{\mathit{Bool}\}[t, f, \mathit{test}] \\
& & \quad \mid t(x, y).\, \overline{y}[].\, 0 \\
& & \quad \mid f(x, y).\, \overline{x}[].\, 0 \\
& & \quad \mid \mathit{test}(b, x, y).\, \overline{b}[y, x].\, 0)
\end{array}
$$

where now *True* signals on its second argument and *False* on its first and the inversion of the behaviors of *True* and *False* is compensated by the fact that *test* reverses the order of its two continuation channels when forwarding them along b. The processes B_1 and B_2 are *not* equivalent under any reasonable untyped notion of equivalence — even the very permissive notion of trace inclusion — because their sets of traces are unrelated. But in the typed setting, they should be considered behaviorally equivalent, since (intuitively) no well-typed observer can distinguish between them. In particular, the polymorphism of the channel *getBools* puts two important constraints on the tests that a well-typed observer may perform on the bodies of B_1 and B_2: that t and f are the only values the observer can send along *test* as first component, and that the observer cannot use t and f as channels. This ensures, for example, that the output along b in B_1 and B_2 cannot be consumed by the observer.

Overview

The main technical contributions of this paper are a definition of behavioral equivalence for polymorphic processes (using a typed version of the notion of *barbed bisimulation* [MS92, San92]) and

an associated proof technique by which equivalences like the one above can be established.[1] The basis of this proof technique is a refinement of the usual *subject reduction* theorem. Given an open process P and an open type environment , for P ("open" in the sense that the process and type environment may have free type variables), the subject reduction theorem shows how to infer constraints on the possible communications that P can perform by examining , (for instance, if a channel appears in , with a completely abstract type, then we can infer that P will not perform communications along this channel). We also rely on a substitution lemma showing that a type substitution does not affect the communication actions that a process can take. To exploit these results on a ground process Q and a ground environment Δ (where "ground" means not containing free type variables) we proceed roughly thus: (1) we take an open process P and an open type environment , in which P is well-typed and of which Q and Δ are ground instances; (2) we use the subject reduction property to infer constraints on the behavior of P; (3) we use the substitution lemma to lift these constraints to Q. The open P and , can be constructed "on the fly," while examining sequences of computation steps beginning from the initial ground processes and type environment (the proofs of our examples follow this schema, exploiting a corollary which directly combines the subject reduction and the substitution results).

The opening sections of the paper (2 to 5) define the syntax, typing rules, and operational semantics of the polymorphic pi-calculus and develop some basic meta-theoretic results, leading up to the extended subject reduction theorem mentioned above. Section 6 defines barbed bisimulation for polymorphic processes and establishes a few useful results. Section 7 then illustrates our proof technique for bisimulation by showing the equivalence of the two implementations of the boolean ADT. Section 8 offers a more ambitious example of our proof technique by verifying the equivalence of two different implementations of symbol tables.

In Section 9, we encounter some surprising results of the interaction between parametricity and aliasing of values. In the presence of aliasing, an abstract data type may turn out to be less abstract than a naive picture of parametric polymorphism might lead one to expect. We show examples in both ML and pi-calculus.

Section 10 surveys related work and briefly explores the prospects for a more "extensional" treatment of parametricity in the polymorphic pi-calculus, following Reynolds's notion of relational parametricity for the polymorphic lambda-calculus. Section 11 discusses some additional issues.

2 Syntax and Notational Preliminaries

We now proceed to formal definitions of the syntax and semantics of the polymorphic pi-calculus. For completeness, we introduce here some additional process combinators not used in the examples in the introduction: the replication construct $!P$, which informally stands for an arbitrary number of copies of P running in parallel; the primitive equality test `if` $x = y$ `then` P `else` Q; and the choice construct $P + Q$, which can behave like either P or Q . The syntax of types and processes

[1] In this paper we have focused on bisimilarity, but we believe that our proof technique can be integrated with other forms of behavioral equivalence like testing and trace equivalence.

is defined as follows:

T	::=	X	type variable
		$\updownarrow\{\widetilde{X}\}[\widetilde{T}]$	polymorphic channel type
P	::=	$x\{\widetilde{X}\}(\widetilde{y}).\,P$	receiver
		$\overline{x}\{\widetilde{T}\}[\widetilde{y}].\,P$	sender
		$(\nu\, x{:}T\,)P$	channel creation
		$P \mid Q$	parallel composition
		0	null process
		$!P$	replication
		$P + Q$	choice
		$\texttt{if}\ x = y\ \texttt{then}\ P\ \texttt{else}\ Q$	equality test

Note that this is a "bare" theoretical calculus, not a programming notation. For example, in a full-scale programming language, we would not want to combine the type constructors for channels, tuples, and type abstractions into a single syntactic form, but would separate them into orthogonal features (cf. [PT96]). They are combined here for technical convenience.

The metavariables P, Q, R are used for process expressions, X, Y, and Z for type variables, S, T, U, and V for types, and lower-case letters for channel names. We abbreviate sequences by writing a tilde over a singleton of the appropriate kind — e.g., $\widetilde{T}$ for a sequence of types — and write $\widetilde{x}{:}\widetilde{T}$ for a sequence of pairs $x_i{:}T_i$ of corresponding elements of $\widetilde{x}$ and $\widetilde{T}$, implicitly assuming that these have the same length. By abuse of notation, operations on singletons are implicitly extended pointwise to sequences; for example, we write $\widetilde{x} \notin \mathrm{dom}(,\,)$ to mean that none of the x_i should be in the domain of the typing context $,\,$.

The type variables $\widetilde{X}$ and the names $\widetilde{y}$ in $x\{\widetilde{X}\}(\widetilde{y}).\,P$ and the name x in $(\nu\, x{:}T\,)P$ are binding occurrences with scope P; the type variables $\widetilde{X}$ in the type $\updownarrow\{\widetilde{X}\}[\widetilde{T}]$ are binding occurrences with scope $\widetilde{T}$. We use alpha-conversion implicitly as necessary to satisfy side conditions about distinctness of bound and free names. The free names of a process P, defined in the obvious way, are written $\mathit{fn}(P)$. We abbreviate a typed channel creation $(\nu\, x{:}T\,)P$ as $(\nu\, x\,)P$ when T is evident or unimportant. When the type component of a polymorphic tuple is empty, we drop it, writing $\overline{s}[t].\,P$ instead of $\overline{s}\{\}[t].\,P$, for example. We often drop 0 as the final suffix of a process expression, writing $\overline{x}[y]$ instead of $\overline{x}[y].\,0$. We write $\prod_{j=1}^{m}$ as abbreviation for $P_1 \mid \ldots \mid P_m$. We assign parallel composition and sum the lowest syntactic precedence among the operators.

We write $[T/X]P$ for the result of substituting the type T for free occurrences of X in P, and similarly $[b/x]$ for substituting channels for channels. Simultaneous substitution is written $[\widetilde{T}/\widetilde{X}]P$. Alpha-conversion is applied silently, as necessary, to avoid capture. The metavariable σ ranges over substitutions of types for type variables. Type substitution is extended pointwise to typing contexts.

A *typing context* $,\,$ is a finite sets of bindings, each mapping a distinct name to a type. We say that a context Δ *extends* a context $,\,$ if $\mathrm{dom}(,\,) \subseteq \mathrm{dom}(\Delta)$ and $,\,(x) = \Delta(x)$ for each $x \in \mathrm{dom}(,\,)$. The set of type variables occurring free in the range of $,\,$ is written $\mathit{TVars}(,\,)$.

A process is said to be *type closed* if it does not contain any free type variables.

3 Typing

A process P is well typed with respect to a typing context $,\,$ if its operations respect the types declared in $,\,$ for its free names. Formally, the *typing relation* $,\, \vdash P$ is the least relation closed

under the following rules:

$$\frac{,\ (a) = \updownarrow\{\widetilde{X}\}[\widetilde{S}] \qquad ,\ (\widetilde{b}) = [\widetilde{T}/\widetilde{X}]\widetilde{S} \qquad ,\ \vdash P}{,\ \vdash \overline{a}\{\widetilde{T}\}[\widetilde{b}].\, P} \tag{T-Out}$$

$$\frac{,\ (a) = \updownarrow\{\widetilde{X}\}[\widetilde{S}] \qquad ,\ ,\ \widetilde{x}{:}\widetilde{S} \vdash P \qquad \widetilde{x} \notin \mathrm{dom}(,\ ) \qquad \widetilde{X} \notin \mathit{TVars}(,\ )}{,\ \vdash a\{\widetilde{X}\}(\widetilde{x}).\, P} \tag{T-In}$$

$$\frac{,\ \vdash P \qquad ,\ \vdash Q}{,\ \vdash P \mid Q} \tag{T-Par}$$

$$\frac{,\ ,\ x{:}\updownarrow\{\widetilde{X}\}[\widetilde{T}] \vdash P \qquad x \notin \mathrm{dom}(,\ )}{,\ \vdash (\nu\, x{:}\updownarrow\{\widetilde{X}\}[\widetilde{T}]\,)P} \tag{T-New}$$

$$\frac{,\ \vdash P}{,\ \vdash\, !P} \tag{T-Repl}$$

$$\frac{,\ \vdash P \qquad ,\ \vdash Q}{,\ \vdash P + Q} \tag{T-Select}$$

$$\frac{,\ (a) = ,\ (b) \qquad ,\ \vdash P \qquad ,\ \vdash Q}{,\ \vdash \mathtt{if}\ a = b\ \mathtt{then}\ P\ \mathtt{else}\ Q} \tag{T-Test}$$

$$,\ \vdash 0 \tag{T-Nil}$$

These rules should be mostly self-explanatory, by analogy with the polymorphic lambda-calculus. In particular, T-Out corresponds to an n-ary variant of the familiar rule of existential introduction

$$\frac{,\ \vdash e \in [T/X]S}{,\ \vdash (\mathit{pack}\ [T, e]\ \mathit{as}\ \exists X.\, S) \in \exists X.\, S}$$

since it packages up a tuple $\widetilde{T}$ of types with a tuple $\widetilde{b}$ of appropriately typed values, hiding some instances of the types $\widetilde{T}$ in the types $\widetilde{S}$ given to the $\widetilde{b}$ by the polymorphic channel a. Similarly, T-In corresponds to the existential elimination rule

$$\frac{,\ \vdash e_1 \in \exists X.\, S \qquad x \notin \mathrm{dom}(,\ ) \qquad X \notin\ \mathit{TVars}(,\ ) \qquad ,\ ,x{:}S \vdash e_2 \in T \qquad X \text{ not free in } T}{,\ \vdash (\mathit{open}\ e_1\ \mathit{as}\ [X, x]\ \mathit{in}\ e_2) \in T}$$

since it unpackages a tuple of types and a tuple of values received along some polymorphic channel and it uses the abstract typing of the received values to typecheck its body. (The final side condition, which prevents a nonsensical escape of the hidden type variable in the lambda-calculus rule, is not needed in the pi-calculus version since the body of a polymorphic input does not directly "yield a value.")

Note that the channel creation operator is only well typed if the type of the new channel is actually a channel type: processes like $(\nu\, x{:}X\,)P$ are not allowed. This prevents a process from creating new (inert) elements of types that it knows only abstractly. On the other hand, we do

allow equality testing between elements of an arbitrary type, since (as we show in Section 9) this kind of testing cannot in general be prevented even in the absence of the testing operator.

Also, note that these rules are *syntax directed*, in the sense that, for each process expression P, there is at most one typing rule that can appear as the final rule in a derivation of , $\vdash P$. This justifies "reading the rules backward" to extract typing derivations for the subexpressions of P from a typing derivation for P itself.

We write Pr_Γ for the set of type-closed processes that are well typed in , .

Technical Properties

We now state some simple static properties of the typing relation that will be needed later. (Here and below, we omit straightforward proofs.)

3.1 Lemma [Weakening]: If , $'$ extends , and , $\vdash P$, then also , $' \vdash P$.

3.2 Lemma [Type substitution preserves typing]: , $\vdash P$ implies σ, $\vdash \sigma P$ for any σ.

3.3 Lemma [Substitution preserves typing]: Suppose that , , $\widetilde{x}{:}\widetilde{S} \vdash R$. If , $'$ extends , and , $'(\widetilde{b}) = \widetilde{S}$, then , $' \vdash [\widetilde{b}/\widetilde{x}]R$.

4 Operational Semantics

We give the operational semantics of processes by means of a labeled transition system, which expresses the internal steps that a process can make and the communications with other processes in which it can engage. The only difference with the standard (early) transition system of the pi-calculus is that, in addition to channels, types may be exchanged in communications. Thus, transitions are of the form $P \xrightarrow{\mu} P'$, where the label μ ranges over *actions* of the following forms:

τ	internal communication
$a\{\widetilde{T}\}[\widetilde{b}]$	input of types $\widetilde{T}$ and values $\widetilde{b}$ at channel a
$(\nu\widetilde{x}{:}\widetilde{S})\overline{a}\{\widetilde{T}\}[\widetilde{b}]$	output of $\widetilde{T}$ and $\widetilde{b}$ at a, extruding bound names $\widetilde{x}$ of type $\widetilde{S}$

In the case of input and output, a is the *subject* of the action. Input and output actions describe possible interactions between P and its environment, while τ actions are placeholders for internal actions in which one subprocess of P communicates with another: an external observer can see that something is happening (time is passing), but nothing more.

The prefix $(\nu\widetilde{x}{:}\widetilde{S})$ in an output action $(\nu\widetilde{x}{:}\widetilde{S})\overline{a}\{\widetilde{T}\}[\widetilde{b}]$ is used to record those names in $\widetilde{b}$ that have been created fresh in P and are not yet known to the environment. (It will always be the case that $\widetilde{x} \subseteq \widetilde{b}$.) When a name b is communicated outside of the scope of the ν that binds it, the ν must be moved outwards to include both the sender and the receiver. Formally, this is accomplished by moving the original ν into the label of the output action (rule R-Open below) and then replacing the ν at the point where the output action meets a corresponding input action and turns into a τ (rule R-Com). This is known as *scope extrusion*.

When an output action has an empty set of extruded names, we drop the ν-part. We write $names(\mu)$ for the set of all channel names appearing in μ, and $bn(\mu)$ for its set of extruded names.

The *labeled transition relation* $P \xrightarrow{\mu} P'$ is defined by the following rules, plus the evident symmetric variants of the rules marked with $*$:

$$a\{\widetilde{X}\}(\widetilde{x}).\, P \xrightarrow{a\{\widetilde{T}\}[\widetilde{b}]} [\widetilde{T}/\widetilde{X}][\widetilde{b}/\widetilde{x}]P \qquad \text{(R-In)}$$

$$\overline{a}\{\widetilde{T}\}[\widetilde{b}].\, P \xrightarrow{\overline{a}\{\widetilde{T}\}[\widetilde{b}]} P \qquad \text{(R-Out)}$$

$$\frac{P \xrightarrow{(\nu \widetilde{x}:\widetilde{S})\overline{a}\{\widetilde{T}\}[\widetilde{b}]} P' \qquad Q \xrightarrow{a\{\widetilde{T}\}[\widetilde{b}]} Q' \qquad \widetilde{x} \notin fn(Q)}{P \mid Q \xrightarrow{\tau} (\nu\, \widetilde{x}:\widetilde{S}\,)(P' \mid Q')} \qquad \text{(R-Com}^*\text{)}$$

$$\frac{P \xrightarrow{\mu} P' \qquad bn(\mu) \cap fn(Q) = \emptyset}{P \mid Q \xrightarrow{\mu} P' \mid Q} \qquad \text{(R-Par}^*\text{)}$$

$$\frac{P \xrightarrow{\mu} P' \qquad x \notin names(\mu)}{(\nu\, x{:}S\,)P \xrightarrow{\mu} (\nu\, x{:}S\,)P'} \qquad \text{(R-New)}$$

$$\frac{P \xrightarrow{(\nu \widetilde{x}:\widetilde{S})\overline{a}\{\widetilde{T}\}[\widetilde{b}]} P' \qquad x \neq a \qquad x \in \{\widetilde{b}\} - \{\widetilde{x}\}}{(\nu\, x{:}S\,)P \xrightarrow{(\nu \widetilde{x}:\widetilde{S}, x:S)\overline{a}\{\widetilde{T}\}[\widetilde{b}]} P'} \qquad \text{(R-Open)}$$

$$\frac{!P \mid P \xrightarrow{\mu} P'}{!P \xrightarrow{\mu} P'} \qquad \text{(R-Repl)}$$

$$\frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'} \qquad \text{(R-Select}^*\text{)}$$

$$\frac{P \xrightarrow{\mu} P'}{\texttt{if } s = s \texttt{ then } P \texttt{ else } Q \xrightarrow{\mu} P'} \qquad \text{(R-Test-T)}$$

$$\frac{s \neq t \qquad Q \xrightarrow{\mu} Q'}{\texttt{if } s = t \texttt{ then } P \texttt{ else } Q \xrightarrow{\mu} Q'} \qquad \text{(R-Test-F)}$$

Note that we are ultimately interested in the operational semantics only of type-closed processes. But we define it for open processes too, since these are needed in order to track different points of view about the types of names. This extension is very mild, as explained by the following easy lemmas.

4.1 Lemma: If P is type-closed and $P \xrightarrow{(\nu \widetilde{x}:\widetilde{S})\overline{a}\{\widetilde{T}\}[\widetilde{b}]} P'$, then $\widetilde{S}$ and $\widetilde{T}$ are ground types.

The two lemmas below show that a type substitution does not affect the possibilities of transitions of processes (this is not true for substitutions of names for names [MPW92]).

4.2 Lemma: $P \xrightarrow{\mu} P'$ implies $\sigma P \xrightarrow{\sigma\mu} \sigma P'$ for any σ.

4.3 Lemma: Suppose that $\sigma P \xrightarrow{\mu} R$.

1. If μ is an output or an internal communication, then there are μ' and P' such that $P \xrightarrow{\mu'} P'$, with $\sigma\mu' = \mu$ and $\sigma P' = R$.

2. If $\mu = a\{\widetilde{T}\}[\widetilde{b}]$, then for any $\widetilde{T'}$ such that $\sigma\widetilde{T'} = \widetilde{T}$ we have $P \xrightarrow{a\{\widetilde{T'}\}[\widetilde{b}]} P'$ for some P' with $\sigma P' = R$.

5 Subject Reduction

In typed calculi, subject reduction expresses the relationship between the operational semantics of a term and a typing for it. In the statement below, the type environment , can be thought as R's "point of view" on the types of its free names. The theorem shows how this point of view evolves under transitions and, most importantly, how it can be used to obtain information about R's possible transitions. For instance, R can only perform input and output actions at names whose type is at least a channel type; and the values sent out by R in an output must satisfy a certain condition on the types. Clause (1), which shows that typing is preserved by internal steps, is the analog of the standard subject reduction property of lambda-calculi (where the operational semantics only talks of "reductions").

5.1 Theorem [Subject reduction]: Suppose , $\vdash R$ and $R \xrightarrow{\mu} R'$, with , , R, and R' possibly open ($\widetilde{S}$ and $\widetilde{T}$ below are also possibly open).

1. If $\mu = \tau$, then , $\vdash R'$.
2. If $\mu = a\{\widetilde{T}\}[\widetilde{b}]$, then, for some $\widetilde{X}$ and $\widetilde{U}$,
   (a) , $(a) = \updownarrow\{\widetilde{X}\}[\widetilde{U}]$,
   (b) if , $'$ extends , and , $'(\widetilde{b}) = [\widetilde{T}/\widetilde{X}]\widetilde{U}$, then , $' \vdash R'$. (Note that some of the $\widetilde{b}$ may already occur in , , while others may be fresh.)
3. If $\mu = (\nu\widetilde{x}{:}\widetilde{S})\overline{a}\{\widetilde{T}\}[\widetilde{b}]$, then, for some $\widetilde{X}$ and $\widetilde{U}$,
   (a) , $(a) = \updownarrow\{\widetilde{X}\}[\widetilde{U}]$,
   (b) $(,\ ,\ \widetilde{x}{:}\widetilde{S})(\widetilde{b}) = [\widetilde{T}/\widetilde{X}]\widetilde{U}$,
   (c) , , $\widetilde{x}{:}\widetilde{S} \vdash R'$,
   (d) each component of $\widetilde{S}$ is a channel type.

Proof: By induction on the length of a derivation of $R \xrightarrow{\mu} R'$, with a case analysis on the last rule used in the derivation. In each case, we implicitly use the fact that the typing rules are syntax directed to read off typing derivations for the subexpressions of a well-typed process expression.

- R-In: We are given

$$\begin{array}{lcl} R & = & a\{\widetilde{X}\}(\widetilde{x}).\,P \\ \mu & = & a\{\widetilde{T}\}[\widetilde{b}] \\ R' & = & [\widetilde{T}/\widetilde{X}][\widetilde{b}/\widetilde{x}]P, \end{array}$$

  from which we must show

  , $(a) = \updownarrow\{\widetilde{X}\}[\widetilde{U}]$
  if , $'$ extends , and , $'(\widetilde{b}) = [\widetilde{T}/\widetilde{X}]\widetilde{U}$, then , $' \vdash [\widetilde{T}/\widetilde{X}][\widetilde{b}/\widetilde{x}]P$.

  The first of these is immediate from T-In, while the second follows from the substitution lemmas (3.3 and 3.2, observing in the second case that $[\widetilde{T}/\widetilde{X}]$, $=$,).

- R-Out: All the required facts are given by the premises to T-Out.

- R-Com: From the premises to R-Com, we have $\mu = \tau$ and $R = P \mid Q$, with $, \vdash P$ and $, \vdash Q$, and
  - $P \xrightarrow{(\nu \widetilde{x}:\widetilde{S})\overline{a}\{\widetilde{T}\}[\widetilde{b}]} P'$, from which the induction hypothesis gives $, , \widetilde{x}:\widetilde{S} \vdash P'$ (using part 3c) and $(, , \widetilde{x}:\widetilde{S})(\widetilde{b}) = [\widetilde{T}/\widetilde{X}]\widetilde{U}$ (using part 3b), and
  - $Q \xrightarrow{a\{\widetilde{T}\}[\widetilde{b}]} Q'$, from which the induction hypothesis gives $, , \widetilde{x}:\widetilde{S} \vdash Q'$ (using part 2b).

  Combining these with T-Com, we obtain $, , \widetilde{x}:\widetilde{S} \vdash P' \mid Q'$, as required.
- R-Par: Case analysis on the form of μ, using the induction hypothesis and weakening.
- R-New: We are given that $R = (\nu\, x{:}S\,)P$ and $R' = (\nu\, x{:}S\,)P'$, with $, , x{:}S \vdash P$ and $P \xrightarrow{\mu} P'$. Proceed by cases on the form of μ.
  - If $\mu = \tau$, then the result follows immediately from the induction hypothesis and T-New.
  - If $\mu = a\{\widetilde{T}\}[\widetilde{b}]$, then the induction hypothesis guarantees that $, (a) = \updownarrow\{\widetilde{X}\}[\widetilde{U}]$ and, for any $, '$ extending $, , x{:}S$, that $, '(\widetilde{b}) = [\widetilde{T}/\widetilde{X}]\widetilde{U}$ implies $, ' \vdash P'$.
    Now, suppose that $, '$ extends $,$ and that $, '(\widetilde{b}) = [\widetilde{T}/\widetilde{X}]\widetilde{U}$. Since x is a bound name, we may assume it is distinct from any name bound by $,$, so $, ', x{:}S$ extends $, , x{:}S$ and we have $, ', x{:}S \vdash P'$. We then obtain $, ' \vdash (\nu\, x{:}S\,)P' = R'$ from T-New.
  - The case where μ is an output is similar.
- R-Open: We are given

  $$\begin{array}{l} R = (\nu\, x{:}S\,)P \\ , , x{:}S \vdash P \\ P \xrightarrow{(\nu \widetilde{x}:\widetilde{S})\overline{a}\{\widetilde{T}\}[\widetilde{b}]} R' \\ \mu = (\nu \widetilde{x}{:}\widetilde{S}, x{:}S)\overline{a}\{\widetilde{T}\}[\widetilde{b}], \end{array}$$

  with $x \neq a$ and $x \in \{\widetilde{b}\} - \{\widetilde{x}\}$. The induction hypothesis gives

  $$\begin{array}{l} (, , x{:}S)(a) = \updownarrow\{\widetilde{X}\}[\widetilde{U}] \\ (, , x{:}S,\ \widetilde{x}{:}\widetilde{S})(\widetilde{b}) = [\widetilde{T}/\widetilde{X}]\widetilde{U} \\ , , x{:}S,\ \widetilde{x}{:}\widetilde{S} \vdash R' \\ \text{each component of } \widetilde{S} \text{ is a channel type.} \end{array}$$

  From this, we obtain the required results as follows: For part (3a), since $x \neq a$, we have $, (a) = (, , x{:}S)(a) = \updownarrow\{\widetilde{X}\}[\widetilde{U}]$. For part (3d), by T-New, S is a channel type, so each component of $S, \widetilde{S}$ is a channel type. For parts (3b) and (3c), simply note that the typing context obtained by extending $, , x{:}S$ with $\widetilde{x}{:}\widetilde{S}$ is identical to the context obtained by extending $,$ with $x{:}S, \widetilde{x}{:}\widetilde{S}$.
- R-Repl, R-Select, R-Test-T, and R-Test-F are straightforward. □

For use in bisimilarity proofs, it is convenient to combine the subject reduction property with the earlier properties of typing and substitution, yielding the corollary below. Intuitively, this corollary says that if P is well typed and σP can perform an interaction, then P itself can perform "the same" interaction and reach a corresponding well-typed state, where the new typing environment is determined by the subject reduction theorem. There are three clauses, corresponding to

the different forms of action that σP might perform (τ, input, output action). Each clause has several conclusions, where the first two use the substitution property to obtain a transition from P corresponding to that of σP and the remaining ones use it to calculate the relationship between P's transition and the original typing , .

5.2 Corollary: Suppose , $\vdash P$ and $\sigma P \xrightarrow{\mu} R$.

1. If $\mu = \tau$, then there is some P' such that
   (a) $P \xrightarrow{\tau} P'$,
   (b) $\sigma P' = R$,
   (c) , $\vdash P'$.
2. If $\mu = a\{\sigma\widetilde{T}\}[\widetilde{b}]$ then, for some $\widetilde{X}$, $\widetilde{U}$, and P',
   (a) $P \xrightarrow{a\{\widetilde{T}\}[\widetilde{b}]} P'$,
   (b) $\sigma P' = R$,
   (c) , $(a) = \updownarrow\{\widetilde{X}\}[\widetilde{U}]$,
   (d) if , $'$ extends , and , $'(\widetilde{b}) = [\widetilde{T}/\widetilde{X}]\widetilde{U}$, then , $' \vdash P'$.
3. If $\mu = (\nu\widetilde{x}{:}\sigma\widetilde{S})\overline{a}\{\sigma\widetilde{T}\}[\widetilde{b}]$ then there are some $\widetilde{X}$, $\widetilde{U}$, and P' such that
   (a) $P \xrightarrow{(\nu\widetilde{x}:\widetilde{S})\overline{a}\{\widetilde{T}\}[\widetilde{b}]} P'$,
   (b) $\sigma P' = R$,
   (c) , $(a) = \updownarrow\{\widetilde{X}\}[\widetilde{U}]$,
   (d) $(, , \widetilde{x}{:}\widetilde{S})(\widetilde{b}) = [\widetilde{T}/\widetilde{X}]\widetilde{U}$,
   (e) , , $\widetilde{x}{:}\widetilde{S} \vdash P'$,
   (f) each component of $\widetilde{S}$ is a channel type.

Proof: By Theorem 5.1 and Lemma 4.3. For instance, in (3), parts (a) and (b) follow from Lemma 4.3(1) and parts (c) to (f) follow from Theorem 5.1. □

6 Bisimulation

We now introduce our basic notions of observational equivalence and develop a few useful properties.

6.1 Barbed bisimulation and equivalence

Barbed bisimulation equates processes that can match each other's interactions and, at each step, can communicate on the same channels. The latter is expressed by means of an observation predicate $\downarrow_a$, for each channel a, that detects the possibility of performing a communication with the external environment along a. That is, $P \downarrow_a$ holds if there are a derivative P' and an action μ with subject a such that $P \xrightarrow{\mu} P'$.

On top of barbed bisimulation, we then define barbed equivalence, which is the behavioral relation we are mainly interested in; here, the requirement on two processes P and Q is that, for all processes R, the compositions $P|R$ and $Q|R$ are barbed bisimilar. In these compositions, R is

thought of as an observer, and the observation predicate $\downarrow_a$ as a signal of success. In CCS and the untyped pi-calculus, barbed equivalence coincides with the ordinary bisimilarities (for the pi-calculus, in the "early" formulation). Of course, in a typed calculus, the processes being compared must obey the same typing and the compositions employed must be compatible with this typing.

6.1.1 Definition: Let Δ be a ground typing. A relation $\mathcal{R} \subseteq Pr_\Delta \times Pr_\Delta$ is a *barbed* Δ-*bisimulation* if $(P, Q) \in \mathcal{R}$ implies:

1. if $P \xrightarrow{\tau} P'$ then there exists Q' such that $Q \xrightarrow{\tau} Q'$ and $(P', Q') \in \mathcal{R}$;
2. if $Q \xrightarrow{\tau} Q'$ then there exists P' such that $P \xrightarrow{\tau} P'$ and $(P', Q') \in \mathcal{R}$;
3. for each channel a, $P \downarrow_a$ iff $Q \downarrow_a$.

Two processes P and Q are said to be *barbed* Δ-*bisimilar*, written $P \dot{\sim}_\Delta Q$, if $(P, Q) \in \mathcal{R}$ for some barbed Δ-bisimulation $\mathcal{R}$.

P and Q are *barbed* Δ-*equivalent*, written $P \sim_\Delta Q$, if, for each ground typing , extending Δ and for each process R such that , $\vdash R$, we have $P|R \dot{\sim}_\Gamma Q|R$.

In the remainder, we write $P \dot{\sim}_\Delta Q$ and $P \sim_\Delta Q$ without recalling that P and Q must be well-typed in Δ and that Δ is ground. The weak version of the equivalences, where one abstracts away from the number of interactions in two matching actions, is obtained in the standard way. Let $\Longrightarrow$ be the reflexive and transitive closure of $\xrightarrow{\tau}$, and let $\Downarrow_a$ be $\Longrightarrow \downarrow_a$, the composition of the two relations. Then *weak barbed* Δ-*bisimulation*, written $\dot{\approx}_\Delta$, is defined by replacing in Definition 6.1.1 the transition $Q \xrightarrow{\tau} Q'$ with $Q \Longrightarrow Q'$ and the predicate $Q \downarrow_a$ with $Q \Downarrow_a$. *Weak barbed* Δ-*equivalence*, written $\approx_\Delta$, is defined by replacing $\dot{\sim}_\Gamma$ with $\dot{\approx}_\Gamma$. The examples in the following sections do not make use of the weak equivalences.

On well-typed processes, our typed barbed equivalences are normally much coarser than the ordinary untyped relations, because the number of legal testers for two processes is smaller: Only those testers that respect the given typing — in particular the constraints imposed by the polymorphic types — are allowed.

6.2 Properties of barbed bisimulation and equivalence

In the bisimilarity clauses of barbed bisimulation, types play no role, because they do not affect interactions and observability of processes. Therefore the standard results on barbed bisimulation in the untyped pi-calculus can be easily adapted to the typed case. In this section, we present a proof technique for typed barbed bisimulation and some simple algebraic laws, and we study the congruence properties of typed barbed equivalence.

6.2.1 Definition: A relation $\mathcal{R} \subseteq Pr_\Delta \times Pr_\Delta$ is a *barbed* Δ-*bisimulation up to* $\dot{\sim}_\Delta$ if $(P, Q) \in \mathcal{R}$ implies:

1. if $P \xrightarrow{\tau} P'$ then there exists Q' such that $Q \xrightarrow{\tau} Q'$ and $P' \dot{\sim}_\Delta \mathcal{R} \dot{\sim}_\Delta Q'$;
2. if $Q \xrightarrow{\tau} Q'$ then there exists P' such that $P \xrightarrow{\tau} P'$ and $P' \dot{\sim}_\Delta \mathcal{R} \dot{\sim}_\Delta Q'$;
3. for each channel a, $P \downarrow_a$ iff $Q \downarrow_a$.

Two processes that are bisimilar up to $\dot{\sim}_\Delta$ are Δ-bisimilar:

6.2.2 Lemma: Suppose that $\mathcal{R}$ is a barbed Δ-bisimulation up to $\dot{\sim}_\Delta$. Then $\mathcal{R} \subseteq \dot{\sim}_\Delta$.

6.2.3 Lemma: The evident laws of commutativity, associativity and absorption of 0 for parallel composition and summation, the unfolding of replication,

$$!P = P|!P,$$

and the extrusion law

$$(\nu\, a{:}T\,)(P|Q) = ((\nu\, a{:}T\,)P)|Q \text{ if } a \text{ is not free in } Q,$$

are all valid for $\sim_\Delta$. (These laws are the main axioms of the structural congruence relation used in "chemical abstract machine style" presentations of the pi-calculus [Mil91].)

6.2.4 Lemma: Suppose that $(\nu\, a{:}T\,)P$ and $(\nu\, a{:}T\,)Q$ are well-typed under Δ. If $P \dot{\sim}_{\Delta,a:T} Q$ then $(\nu\, a{:}T\,)P \dot{\sim}_\Delta (\nu\, a{:}T\,)Q$.

Typed barbed equivalence enjoies the same kind of congruence properties as ordinary labeled bisimulation of the untyped pi-calculus [MPW92].

6.2.5 Lemma: If $P \sim_{\Delta,a:T} Q$ then $(\nu\, a{:}T\,)P \sim_\Delta (\nu\, a{:}T\,)Q$.

Proof: A consequence of Lemma 6.2.4 and of the extrusion law of Lemma 6.2.3. □

6.2.6 Lemma: If $P \sim_\Delta Q$ then $!P \sim_\Delta !Q$.

Proof: A simple diagram chase, exploiting the technique of bisimulation up-to $\dot{\sim}_\Delta$. □

6.2.7 Lemma: If $P \sim_\Delta Q$ and $\Delta \vdash R$ then also $P + R \sim_\Delta Q + R$.

6.2.8 Lemma: Suppose that $P \sim_\Delta Q$, that $\Delta(t) = \Delta(s)$, and that $\Delta \vdash R$. Then it holds that

$$\texttt{if } s = t \texttt{ then } P \texttt{ else } R \;\sim_\Delta\; \texttt{if } s = t \texttt{ then } Q \texttt{ else } R$$

and

$$\texttt{if } s = t \texttt{ then } R \texttt{ else } P \;\sim_\Delta\; \texttt{if } s = t \texttt{ then } R \texttt{ else } Q.$$

The relation $\sim_\Delta$ is also preserved by output prefix, and, by definition, by parallel composition. As usual for pi-calculus bisimilarities, congruence fails for input prefix. The same congruence properties hold in the case of *weak* Δ-barbed equivalence ($\approx_\Delta$) except, as usual for bisimilarity, for the congruence with respect to summation.

7 Example: Boolean ADTs

We now show that the two implementations of booleans from the introduction are behaviorally indistinguishable when the constraints imposed by the polymorphic types are taken into account.

Let , $\stackrel{\text{def}}{=} \mathit{getBools} : \updownarrow\{X\}[X, X, \updownarrow[X, \updownarrow[\,], \updownarrow[\,]]]$. To show that $B_1 \sim_\Gamma B_2$, we have to prove $B_1 \mid R \dot{\sim}_\Delta B_2 \mid R$ for any ground Δ extending , and any ground R such that $\Delta \vdash R$. Let

$$\begin{array}{rcllrcl}
T_1 & \stackrel{\text{def}}{=} & t(x,y).\,\overline{x}[\,] & & T_2 & \stackrel{\text{def}}{=} & t(x,y).\,\overline{y}[\,] \\
F_1 & \stackrel{\text{def}}{=} & f(x,y).\,\overline{y}[\,] & & F_2 & \stackrel{\text{def}}{=} & f(x,y).\,\overline{x}[\,] \\
IF_1 & \stackrel{\text{def}}{=} & \mathit{test}(b,x,y).\,\overline{b}[x,y] & & IF_2 & \stackrel{\text{def}}{=} & \mathit{test}(b,x,y).\,\overline{b}[y,x].
\end{array}$$

Using these abbreviations, the definitions of B_1 and B_2 become:

$$\begin{array}{lll} B_1 & \stackrel{\text{def}}{=} & (\nu\, t{:}Bool,\ f{:}Bool,\ test{:}\updownarrow[\,Bool, \updownarrow[\,], \updownarrow[\,]\,]) \\ & & \quad (\overline{getBools}\{Bool\}[t, f, test] \mid T_1 \mid F_1 \mid IF_1) \end{array}$$

$$\begin{array}{lll} B_2 & \stackrel{\text{def}}{=} & (\nu\, t{:}Bool,\ f{:}Bool,\ test{:}\updownarrow[\,Bool, \updownarrow[\,], \updownarrow[\,]\,]) \\ & & \quad (\overline{getBools}\{Bool\}[t, f, test] \mid T_2 \mid F_2 \mid IF_2) \end{array}$$

We now verify that the union of the following sets $\mathcal{R}_i$ of pairs of processes is a barbed Δ-bisimulation up to $\dot{\sim}_\Delta$. We only check clause (1) of the definition of barbed bisimulation, since clause (2) is similar to (1) and clause (3) is straightforward.

- $\mathcal{R}_1$ has all pairs of the form $(B_1 \mid R, B_2 \mid R)$ such that $\Delta \vdash R$.
- $\mathcal{R}_2$ has all pairs of the form

$$\left(\begin{array}{l} (\nu\, t, f, test)(T_1 \mid F_1 \mid IF_1 \mid [Bool/X]R), \\ (\nu\, t, f, test)(T_2 \mid F_2 \mid IF_2 \mid [Bool/X]R) \end{array}\right)$$

  for R such that

$$\Delta, t : X, f : X, test : \updownarrow[\,X, \updownarrow[\,], \updownarrow[\,]\,] \vdash R.$$

- $\mathcal{R}_3$ has all pairs of the form

$$\left(\begin{array}{l} (\nu\, t, f, test, \widetilde{p} : \updownarrow[\,]\,)(T_1 \mid F_1 \mid \overline{h}[c, d] \mid [Bool/X]R), \\ (\nu\, t, f, test, \widetilde{p} : \updownarrow[\,]\,)(T_2 \mid F_2 \mid \overline{h}[d, c] \mid [Bool/X]R) \end{array}\right)$$

  for $\widetilde{p}, R$ such that

$$\begin{array}{l} \Delta, t : X, f : X, test : \updownarrow[\,X, \updownarrow[\,], \updownarrow[\,]\,], \widetilde{p} : \updownarrow[\,] \vdash R \\ \widetilde{p} \subseteq \{c, d\} \\ h \in \{t, f\}. \end{array}$$

- $\mathcal{R}_4$ has all pairs of the form

$$\left(\begin{array}{l} (\nu\, t, f, test, \widetilde{p} : \updownarrow[\,]\,)(N_1 \mid \overline{c}[] \mid [Bool/X]R), \\ (\nu\, t, f, test, \widetilde{p} : \updownarrow[\,]\,)(N_2 \mid \overline{c}[] \mid [Bool/X]R) \end{array}\right)$$

  for $\widetilde{p}, R, N_1, N_2$ such that

$$\begin{array}{l} \Delta, t : X, f : X, test : \updownarrow\{\}[X, \updownarrow[\,], \updownarrow[\,]], \widetilde{p} : \updownarrow[\,] \vdash R \\ \widetilde{p} \subseteq \{c\} \\ \{N_1, N_2\} \subseteq \{T_1, F_1, T_2, F_2\} \end{array}$$

- $\mathcal{R}_5$ has all pairs of the form

$$\left(\begin{array}{l}(\nu\, t, f, test)(N1 \mid [Bool/X]R),\\ (\nu\, t, f, test)(N2 \mid [Bool/X]R)\end{array}\right)$$

for R, N_1, N_2 such that

$$\Delta, t : X, f : X, test : \updownarrow[\,X, \updownarrow[\,\,], \updownarrow[\,\,]\,] \vdash R$$
$$\{N_1, N_2\} \subseteq \{T_1, F_1, T_2, F_2\}.$$

These sets are constructed so that each pair of processes in $\mathcal{R}_i$ can match each other's interactions with the derivatives forming pairs of processes that are either in $\mathcal{R}_i$ or in $\mathcal{R}_{i+1}$. In the case of $\mathcal{R}_1$, the interesting case is the interaction between B_1 and R, where B_1 makes the output at *getBools* and R the input. By Corollary 5.2(2), one can infer that the input from R is of the form

$$R \xrightarrow{getBools\{\sigma X\}[t,f,test]} \sigma R'$$

where $\sigma = [Bool/X]$ and R' satisfies the side conditions in the definition of $\mathcal{R}_2$. Process $B_2 \mid R$ matches this interaction in the similar way.

We now show in detail the argument for $\mathcal{R}_2$ (the argument for the rest of the $\mathcal{R}_i$ is similar or easier). Suppose the process

$$(\nu\, t, f, test)(T_1 \mid F_1 \mid IF_1 \mid [Bool/X]R)$$

has an interaction. If only the subprocess $[Bool/X]R$ contributes to the action, then, by Corollary 5.2, its move can be written as $[Bool/X]R \xrightarrow{\tau} [Bool/X]R'$ where R' is well typed under the same typing as R. In this case, the process $(\nu\, t, f, test)(T_2 \mid F_2 \mid IF_2 \mid [Bool/X]R)$ can make a matching step, and the two derivatives are again in $\mathcal{R}_2$.

By definition, no interaction is possible within the system $T_1 \mid F_1 \mid IF_1$, so it remains only to consider the case of an interaction in which both $T_1 \mid F_1 \mid IF_1$ and $[Bool/X]R$ take part. Process R is well-typed under the assumptions $t : X, f : X, test : \updownarrow[\,X, \updownarrow[\,\,], \updownarrow[\,\,]\,]$. Since the type of t and f is not a channel type, by Corollary 5.2 $[Bool/X]R$ cannot perform visible actions with t or f as subject. Therefore the only possible interactions between $T_1 \mid F_1 \mid IF_1$ and $[Bool/X]R$ are along the channel *test*. In this case, $[Bool/X]R$ contributes an output. Moreover, since for R the first argument in the type of *test* is X, by Corollary 5.2(clauses 3d,f) any output at *test* by $[Bool/X]R$ will have either t or f as first argument (the appeal to clause (f) of Corollary 5.2(3) is needed to exclude the case in which this argument is a fresh channel). Suppose it is t (the other case is symmetric). Then the output from $[Bool/X]R$ is

$$[Bool/X]R \xrightarrow{(\nu \widetilde{p}:\updownarrow[\,\,])\,\overline{test}\,[t,c,d]} [Bool/X]R',$$

where c and d have type $\updownarrow[\,\,]$ and $\widetilde{p} \subseteq \{c, d\}$. Thus, up to the laws of Lemma 6.2.3, the action is

$$\begin{array}{r}(\nu\, t, f, test)(T_1 \mid F_1 \mid IF_1 \mid [Bool/X]R) \quad \xrightarrow{\tau}\\ (\nu\, t, f, test, \widetilde{p} : \updownarrow[\,\,])(T_1 \mid F_1 \mid \overline{t}[c, d] \mid [Bool/X]R').\end{array}$$

This is matched (up to the laws of Lemma 6.2.3) by the action

$$\begin{array}{r}(\nu\, t, f, test)(T_2 \mid F_2 \mid IF_2 \mid [Bool/X]R) \quad \xrightarrow{\tau}\\ (\nu\, t, f, test, \widetilde{p} : \updownarrow[\,\,])(T_2 \mid F_2 \mid \overline{t}[d, c] \mid [Bool/X]R'),\end{array}$$

since, by Corollary 5.2(3e), we have

$$\Delta, t : X, f : X, test : \updownarrow[\, X, \updownarrow[\,], \updownarrow[\,]\,], \widetilde{p} : \updownarrow[\,] \vdash R'$$

and therefore the side condition in the definition of $\mathcal{R}_3$ is satisfied. This completes the argument.

Another interesting example is obtained replacing, in B_1, the line implementing the conditional test with:

$$test(b, x, y).\ \texttt{if}\ (b = t) \vee (b = f)\ \texttt{then}\ \overline{b}[x, y]\ \texttt{else}\ BAD$$

where BAD can be any process. This new package is equivalent to B_1 because the value received at *test* for b is always either t or f. This example shows that a client of the ADT is not authorized to make up new values of type *Bool*, since the client knows nothing about this type.

None of these equivalences hold for the ordinary untyped pi-calculus, because without typing we cannot impose appropriate constraints on the actions that an observer can make. For instance, there are several traces that break the trace equivalence between B_1 and B_2: e.g.,

$$\overline{getBools}\{Bool\}[t, f, test].\ test[t, x, y].\overline{t}[x, y]$$

is a trace of B_1 but not of B_2.

8 Example: Two Implementations of a Symbol Table Package

We now apply our proof techniques to a more challenging example: two implementations of a symbol table package. The two table implementations use quite different representations for the keys (strings vs. integers) and different implementations of the package operations, hence they have quite different untyped behaviors.

A symbol table stores an association of (a finite set of) strings to values of some type which is unknown outside — the abstract type of *keys*. The only operation that clients can perform on keys is to use the table to compare them for equality. The client may also insert a string in the table, in which case an appropriate key is returned.

The client uses a channel *getST* to obtain the channels for making insertion and equality-test requests. As in the boolean example, *getST* is polymorphic, hiding the concrete type of keys. The two tables use different implementations for this concrete type. In one case, the type is strings and the association function is a partial identity function. In the second case, the type is integers and the association function is a partial injective function from strings to integers.

To make the examples more readable, we use an extended process syntax including communication of integers and strings, union- and membership-test operations on sets, and recursive process definitions. These constructs could be taken as syntactic sugar, since data values and recursive definitions can be coded in the pi-calculus [Mil91]; for brevity of the following proofs, however, we shall take them as extensions of the syntax, since their meaning is clear and they can be accommodated in our theory with only minor and obvious modifications.

The main bodies of the two table implementations (ST_1 and ST_2) are the recursive processes $Loop_1$ and $Loop_2\langle B, n\rangle$; in the latter the two parameters are a finite set B of pairs of strings and integers (giving the association function of the table) and a counter n that stores the first integer not used in B. An insertion request has two parameters: a string t and a return channel r. Process $Loop_1$ simply returns a reference to t. Process $Loop_2\langle B, n\rangle$ returns a reference to the integer associated with string t, if an entry for t in B exists; otherwise it returns a reference to n, adds the pair (t, n) to the set B, and increments the counter. An equality-test request has four

parameters v_1, v_2, f, g; the table returns an answer at f or g, depending on whether the the values referenced to by v_1 and v_2 are equal or not.

We define the two symbol tables ST_1 and ST_2. They will be well typed under the typing

$$, \stackrel{\text{def}}{=} getST : \updownarrow\{X\}[\updownarrow[\,\texttt{String}, \updownarrow[\,X\,]\,], \updownarrow[\,X, X, \updownarrow[\,], \updownarrow[\,]\,]].$$

Below, B ranges over sets of pairs of strings and integers; t and s over strings, and n, m, i, and j over integers; other words in lowercase letters are channels. We write S for the type $\updownarrow[\,\texttt{String}\,]$ and T for $\updownarrow[\,\texttt{Int}\,]$.

$$\begin{array}{lcl}
ST_1 & \stackrel{\text{def}}{=} & \nu(ins{:}\updownarrow[\,\texttt{String}, \updownarrow[\,S\,]\,], \\
& & \quad eq{:}\updownarrow[\,\texttt{String}, \texttt{String}, \updownarrow[\,], \updownarrow[\,]\,]) \\
& & \quad\quad (\overline{getST}\{\updownarrow[\,\texttt{String}\,]\}[ins, eq] \\
& & \quad\quad \mid Loop_1) \\
ST_2 & \stackrel{\text{def}}{=} & \nu(ins{:}\updownarrow[\,\texttt{String}, \updownarrow[\,T\,]\,], \\
& & \quad eq{:}\updownarrow[\,\texttt{Int}, \texttt{Int}, \updownarrow[\,], \updownarrow[\,]\,]) \\
& & \quad\quad (\overline{getST}\{\updownarrow[\,Int\,]\}[ins, eq] \\
& & \quad\quad \mid Loop_2\langle\emptyset, 0\rangle)
\end{array}$$

where $Loop_1$ and $Loop_2\langle B, n\rangle$ are defined as follows:

$$\begin{array}{lcl}
Loop_1 & \stackrel{\text{def}}{=} & \\
& & \quad eq(v_1, v_2, f, g).\, v_1(h_1).\, v_2(h_2). \\
& & \quad\quad \texttt{if}\ h_1 = h_2 \\
& & \quad\quad \texttt{then}\ \overline{f}[].\, Loop_1 \\
& & \quad\quad \texttt{else}\ \overline{g}[].\, Loop_1 \\
& + & \quad ins(t, r).\, (\nu\, u{:}S\,)(\overline{r}[u].\, Loop_1 \mid\, !\overline{u}[t])
\end{array}$$

$$\begin{array}{lcl}
Loop_2\langle B, n\rangle & \stackrel{\text{def}}{=} & \\
& & \quad eq(v_1, v_2, f, g).\, v_1(h_1).\, v_2(h_2). \\
& & \quad\quad \texttt{if}\ h_1 = h_2 \\
& & \quad\quad \texttt{then}\ \overline{f}[].\, Loop_2\langle B, n\rangle \\
& & \quad\quad \texttt{else}\ \overline{g}[].\, Loop_2\langle B, n\rangle \\
& + & \quad ins(t, r). \\
& & \quad\quad \texttt{if}\ \exists\ i \text{ such that } (t, i) \in B \\
& & \quad\quad \texttt{then}\ (\nu\, u{:}T\,)(\overline{r}[u].\, Loop_2\langle B, n\rangle \mid\, !\overline{u}[i]) \\
& & \quad\quad \texttt{else}\ (\nu\, u{:}T\,)(\overline{r}[u].\, Loop_2\langle B \cup \{(t, n)\}, n{+}1\rangle \mid\, !\overline{u}[n])
\end{array}$$

Note that, in both tables, the values sent back to the client after an insertion are not actual values of the abstract type (integers or strings), but references to them. This is to "protect" the values, and is important for the proof of bisimilarity. We discuss this point further in Section 9.

To show that $ST_1 \sim_\Gamma ST_2$, we have to prove that $ST_1 \mid R \,\dot{\sim}_\Delta\, ST_2 \mid R$ for all Δ extending the type environment , and all R such that $\Delta \vdash R$. We define a barbed Δ-bisimulation $\mathcal{R}$ up to $\dot{\sim}_\Delta$ as the union of the following sets $\mathcal{R}_i$ of pairs of processes:

- $\mathcal{R}_1$ contains all pairs of the form $(ST_1 \mid R,\ ST_2 \mid R)$ such that $\Delta \vdash R$.

- $\mathcal{R}_2$ contains all pairs of the form

$$\left(\begin{array}{l}(\nu\, ins, eq, u_1, \ldots, u_m)\\ \quad (Loop_1 \mid \prod_{j=1}^{m} !\overline{u_j}[t_j] \mid [S/X]R),\\ (\nu\, ins, eq, u_1, \ldots, u_m)\\ \quad (Loop_2\langle B, n\rangle \mid \prod_{j=1}^{m} !\overline{u_j}[n_j] \mid [T/X]R)\end{array}\right)$$

where

$$B \stackrel{\text{def}}{=} \{(s_i, i) \;:\; 0 \le i < n\} \tag{1}$$

subject to the conditions

$$\Delta, ins : \updownarrow[\,\texttt{String}, \updownarrow[\,X\,]\,], eq : \updownarrow[\,X, X, \updownarrow[\,], \updownarrow[\,]\,], u_1 : X, \ldots, u_m : X \vdash R \tag{2}$$

$$m, n \ge 0 \tag{3}$$

$$\{s_i \;:\; 0 \le i < n\} = \{t_j \;:\; 1 \le j \le m\} \tag{4}$$

$$\text{for all } 1 \le j \le m \text{ it holds that } 0 \le n_j < n \tag{5}$$

$$\text{for all } 1 \le j_1, j_2 \le m \text{ it holds that } t_{j_1} = t_{j_2} \text{ iff } n_{j_1} = n_{j_2} \tag{6}$$

(Condition (4) ensures that the sets of strings which have been inserted into the two tables are the same. Condition 5 ensures that the integer keys used in the second table do not exceed the parameter n of $Loop_2\langle B, n\rangle$. Condition 6 ensures that the two tables agree on the equalities of keys stored in corresponding positions.)

- $\mathcal{R}_3$ contains all pairs of the form

$$\left(\begin{array}{l}(\nu\, ins, eq, u_1, \ldots, u_m, \widetilde{p})\\ \quad (P_1 \mid \prod_{j=1}^{m} !\overline{u_j}[t_j] \mid [S/X]R),\\ (\nu\, ins, eq, u_1, \ldots, u_m, \widetilde{p})\\ \quad (P_2 \mid \prod_{j=1}^{m} !\overline{u_j}[n_j] \mid [T/X]R)\end{array}\right)$$

for

$$\begin{array}{lcl} P_1 & \stackrel{\text{def}}{=} & u_{j_1}(h_1).\, u_{j_2}(h_2).\\ & & \quad \texttt{if } h_1 = h_2\\ & & \quad \texttt{then } \overline{f}[].\, Loop_1\\ & & \quad \texttt{else } \overline{g}[].\, Loop_1\\ P_2 & \stackrel{\text{def}}{=} & u_{j_1}(h_1).\, u_{j_2}(h_2).\\ & & \quad \texttt{if } h_1 = h_2\\ & & \quad \texttt{then } \overline{f}[].\, Loop_2\langle B, n\rangle\\ & & \quad \texttt{else } \overline{g}[].\, Loop_2\langle B, n\rangle \end{array}$$

with B defined as in (1) and subject to the conditions (3)-(6) plus

$$\begin{array}{l}\Delta, \mathit{ins} : \updownarrow[\,\mathtt{String}, \updownarrow[\,X\,]\,], \mathit{eq} : \updownarrow[\,X, X, \updownarrow[\,], \updownarrow[\,]\,], \\ u_1 : X, \ldots, u_m : X, \widetilde{p} : \updownarrow[\,] \vdash R\end{array} \tag{7}$$

and

$$\begin{array}{l}1 \leq j_1, j_2 \leq m \\ \widetilde{p} \subseteq \{f, g\}.\end{array}$$

- $\mathcal{R}_4$ is defined in the same way as $\mathcal{R}_3$, except that P_1 and P_2 are now defined like this:

$$\begin{array}{lcl} P_1 & \stackrel{\text{def}}{=} & u_{j_2}(h_2).\,\mathtt{if}\ t_{j_1} = h_2 \\ & & \quad \mathtt{then}\ \overline{f}[].\,\mathit{Loop}_1 \\ & & \quad \mathtt{else}\ \overline{g}[].\,\mathit{Loop}_1 \\ P_2 & \stackrel{\text{def}}{=} & u_{j_2}(h_2).\,\mathtt{if}\ n_{j_1} = h_2 \\ & & \quad \mathtt{then}\ \overline{f}[].\,\mathit{Loop}_2\langle B, n\rangle \\ & & \quad \mathtt{else}\ \overline{g}[].\,\mathit{Loop}_2\langle B, n\rangle \end{array}$$

- $\mathcal{R}_5$ is defined in the same way as $\mathcal{R}_3$, except that P_1 and P_2 are now defined like this:

$$\begin{array}{lcl} P_1 & \stackrel{\text{def}}{=} & \mathtt{if}\ t_{j_1} = t_{j_2}\ \mathtt{then}\ \overline{f}[].\,\mathit{Loop}_1\ \mathtt{else}\ \overline{g}[].\,\mathit{Loop}_1 \\ P_2 & \stackrel{\text{def}}{=} & \mathtt{if}\ n_{j_1} = n_{j_2} \\ & & \quad \mathtt{then}\ \overline{f}[].\,\mathit{Loop}_2\langle B, n\rangle \\ & & \quad \mathtt{else}\ \overline{g}[].\,\mathit{Loop}_2\langle B, n\rangle \end{array}$$

- $\mathcal{R}_6$ contains all pairs of the form

$$\left(\begin{array}{l} (\nu\, \mathit{ins}, \mathit{eq}, u_1, \ldots, u_m, \widetilde{p}\,) \\ \quad (P_1 \mid \prod_{j=1}^{m} !\overline{u_j}[t_j] \mid [S/X]R), \\ (\nu\, \mathit{ins}, \mathit{eq}, u_1, \ldots, u_m, \widetilde{p}\,) \\ \quad (P_2 \mid \prod_{j=1}^{m} !\overline{u_j}[n_j] \mid [T/X]R) \end{array}\right)$$

for

$$\begin{array}{lcl} P_1 & \stackrel{\text{def}}{=} & (\nu\, u{:}S\,)(\overline{r}[u].\,\mathit{Loop}_1 \mid !\overline{u}[t]) \\ P_2 & \stackrel{\text{def}}{=} & \mathtt{if}\ (t, i) \in B \\ & & \quad \mathtt{then}\ (\nu\, u{:}T\,)(\overline{r}[u].\,\mathit{Loop}_2\langle B, n\rangle \mid !\overline{u}[i]) \\ & & \quad \mathtt{else}\ (\nu\, u{:}T\,) \\ & & \qquad (\overline{r}[u].\,\mathit{Loop}_2\langle B \cup (t, n), n{+}1\rangle \mid !\overline{u}[n]) \end{array}$$

with B defined as in (1) and subject to conditions (3)-(6) plus

$$\begin{array}{l}\Delta, \mathit{ins} : \updownarrow[\,\mathtt{String}, \updownarrow[\,X\,]\,], \mathit{eq} : \updownarrow[\,X, X, \updownarrow[\,], \updownarrow[\,]\,], \\ u_1 : X, \ldots, u_m : X, \widetilde{p} : \updownarrow[\,X\,] \vdash R\end{array}$$

and

$$\widetilde{p} \subseteq \{r\}.$$

For each $\mathcal{R}_i$, we have to show that the processes in the pairs in $\mathcal{R}_i$ can match each other's actions. We consider the actions of the first process, and sketch the proof for the main cases (again checking only clause (1) of the definition of barbed bisimulation). We elide applications of the laws in Lemma 6.2.3.

$\mathcal{R}_1$: Proceed as for the pairs of $\mathcal{R}_1$ in the boolean package example of Section 7. In the case of interaction between, on the one hand, ST_1 and R and, on the other hand, ST_2 and R, the pair of derivatives is $\big((\nu\, ins, eq\,)(Loop_1 \mid [S/X]R'), (\nu\, ins, eq\,)(Loop_2\langle\emptyset, 0\rangle \mid [T/X]R')\big)$, for some R', and it is in $\mathcal{R}_2$.

$\mathcal{R}_2$: From (2) and Corollary 5.2(2c) we know that the process $[S/X]R$ cannot interact at a channel u_j. There can be interactions between $[S/X]R$ and $Loop_1$ along channels eq and ins. From (2) and Corollary 5.2 the action performed by R is either

$$R \xrightarrow{(\nu \widetilde{p}:\updownarrow[\,])\,\overline{eq}\,[u_{j_1}, u_{j_2}, f, g]} R'$$

with

$$\begin{array}{l} \Delta, ins : \updownarrow[\,\mathtt{String}, \updownarrow[\,X\,]\,], eq : \updownarrow[\,X, X, \updownarrow[\,], \updownarrow[\,]\,], \\ \quad u_1 : X, \ldots, u_m : X, \widetilde{p} : \updownarrow[\,] \vdash R' \\ 1 \le j_1, j_2 \le m \\ \widetilde{p} \subseteq \{f, g\} \end{array} \tag{8}$$

or

$$R \xrightarrow{(\nu \widetilde{p}:\updownarrow[\,X\,])\,\overline{ins}\,[t,r]} R'$$

with

$$\begin{array}{l} \Delta, ins : \updownarrow[\,\mathtt{String}, \updownarrow[\,X\,]\,], eq : \updownarrow[\,X, X, \updownarrow[\,], \updownarrow[\,]\,], \\ \quad u_1 : X, \ldots, u_m : X, \widetilde{p} : \updownarrow[\,X\,] \vdash R' \\ \widetilde{p} \subseteq \{r\}. \end{array}$$

In the former case, the interaction between $[S/X]R$ and $Loop_1$ can be matched by an interaction between $[T/X]R$ and $Loop_2\langle B, n\rangle$, and the two derivatives form a pair of processes in $\mathcal{R}_3$.

In the latter case, the interactions of $[S/X]R$ and $[T/X]R$ with, respectively, $Loop_1$ and $Loop_2\langle B, n\rangle$ produce pairs of derivatives in $\mathcal{R}_6$.

$\mathcal{R}_3$: Because of (7) and Corollary 5.2, process $[S/X]R$ cannot perform actions at a channel u_j. There can be interactions between P_1 and process $!\overline{u_{j_1}}[t_{j_1}]$. The analogous interaction between P_2 and $!\overline{u_{j_1}}[n_{j_1}]$ yields a pair of processes in $\mathcal{R}_4$.

$\mathcal{R}_4$: Reason as for $\mathcal{R}_3$. Interactions between, on the one hand, P_1 and $!\overline{u_{j_2}}[t_{j_2}]$, and, on the other hand, P_2 and $!\overline{u_{j_2}}[n_{j_2}]$, give a pair of processes in $\mathcal{R}_5$.

$\mathcal{R}_5$: The additional element to note is that by (6), $n_{j_1} = n_{j_2}$ iff $t_{j_1} = t_{j_2}$. The pair of derivatives is in $\mathcal{R}_2$.

$\mathcal{R}_6$: We must distinguish between the case when there is $0 \le i < n$ such that $(t, i) \in B$ and the case when there is no such i. In either cases, interactions, on the one hand, between P_1 and $[S/X]R$, and, on the other hand, between P_2 and $[T/X]R$, give a pair of processes in $\mathcal{R}_2$.

9 Aliasing and Information Leakage

In the language considered in this paper, we have allowed conditional operators at arbitrary types: a process can always test for equality or inequality between two values of the same type. When the process's knowledge of the type of the two values is partial, this permits a "leakage of information" that gives receivers of polymorphic communications some unexpected discriminating power. For instance, suppose that x is a channel of type $\updownarrow\{X\}[X, X, \updownarrow[\,X\,]]$. We intuitively expect that the recipient of a triple of values $[a, b, c]$ sent along x should not be able to do anything with the a and b except to send them back along c. But the conditional operator also allows recipients to test whether a and b are equal. For another example, consider a variant $ST1'$ of the symbol table package $ST1$ of Section 8 where, on insertions of the same string, the same reference is returned. An observer can distinguish $ST1$ from $ST1'$, because the values that the former returns after an insert are always different, whereas those returned by the latter may be equal and therefore may enable some matching. To protect against this leakage, in the two symbol table implementations we have adopted a "safe" programming style in which all values transmitted abstractly to the outside world are protected by fresh channels; this makes the equality test available to an observer useless.

Unfortunately (and, to us, rather surprisingly), this kind of testing by the receiver cannot be prevented in general, in the sense that it can sometimes be simulated even in a language without *any* conditional operator. Returning to the first example, for instance, suppose that, in addition to x, there is a global channel g of type $\updownarrow[\updownarrow[\,]\,]$, and consider the receiver process

$$P \stackrel{\text{def}}{=} x\{X\}(m, n, o).\,(\overline{o}[m].\,\overline{o}[n] \mid g(i).\,g(j).\,(\overline{i}[] \mid j().\,\overline{equal}[])). \tag{9}$$

Having received m, n, and o along x, P sends m and n along o (the only well-typed thing it can do with m and n) and, in parallel, listens at g for two channels i and j, which it then tests by sending a signal on i and listening to see whether it is received on j, emitting a success signal on a global channel *equal* if so. We can say that P tests m and n for equality, in the sense that if we send it the tuple $\{\updownarrow[\,]\}[a, a, g]$ along x (assuming $a : \updownarrow[\,]$), then it can emit a signal on *equal*, whereas if we send it $\{\updownarrow[\,]\}[a, b, c]$ along x (where $a \neq b$ or $c \neq g$), it cannot emit a signal on *equal*.

In this example, information leakage allows a process to detect the identity of names whose type is abstract. In general, due to leakage, a process may succeed in using a value with a capability that is not part of the type with which the value had been received in an input. Here is another example. Suppose y has type $\updownarrow\{X\}[\updownarrow[\,X\,], X]$, and that a process Q receives two new names b and c at y:

$$Q \xrightarrow{y\{\texttt{Int}\}[b,c]} Q_1$$

Because of the type of y, the only interesting capability on b and c received by Q_1 is to carry the latter along the former. In particular, Q_1 does not receive the capability of performing actions at c. Suppose now that a channel z has type $\updownarrow[\updownarrow[\updownarrow[\texttt{Int}\,]\,]\,]$ and that Q_1, in an input at z, receives b again:

$$Q_1 \xrightarrow{z\{\}[b]} Q_2$$

Since b is received over a monomorphic channel, the knowledge of the type of b improves. We might naively expect that the improvement does not affect the capabilities on c, since c is not mentioned in the action at all. But this is not true, for instance, if $Q_2 \stackrel{\text{def}}{=} \overline{b}[c] \mid b(x).\,\overline{x}[5]$: after the interaction $Q_2 \xrightarrow{\tau} \overline{c}[5]$, name c is used with the concrete type $\updownarrow[\,\texttt{Int}\,]$, showing that the process has acquired the capability of using c as a channel.

Leakage is caused by aliasing — the fact that different variables in the text of a process can be instantiated to the same channel value, say b. The variables may have different types and, as a consequence, the process may succeed in using the union of the capabilities provided by these types on b; moreover the increase may affect the capabilities on channels which had been received together with b in an input — like c in the previous example. (This also explains why, in our formulation of the subject reduction theorem, typing environments have *unique* binding for channels, as opposed to, say, typing environments with multiple binding for channels so to track the possible difference in the type with which distinct occurrences of a channel in a process have been created.)

The real significance of these examples of information leakage is not at present clear to us. Nor is it clear whether they can be avoided, e.g., by identifying syntactic or typing restrictions on processes that would guarantee that information leakage cannot occur. For example, we cannot just forbid aliasing of names passed with completely abstract types, since in example (9) it is o, not m or n, that is aliased. Moreover, in (9) it would not even be enough to require that the third name passed to P along x should not be aliased, since it is easy to construct variants of P where the concrete reference to g is not a global channel but is obtained from the outside world by a later communication.

Information leakage is not peculiar to the pi-calculus: similar examples can be constructed in any setting with both polymorphism and aliasing. For instance, for an example similar to (9) in Standard ML, let the global variable `g` be an integer reference cell

```
val g = ref 0
      : int ref
```

and consider the following function `f`:

```
fun f r m n = (g:=0; r:=m;
               let val i = !g in
                 g:=1; r:=n; (!g = i)
               end)
            : 'a ref -> 'a -> 'a -> bool
```

Then

$$\texttt{f r x y} = \begin{cases} \texttt{true} & \text{if } \texttt{x} = \texttt{y} \text{ and } \texttt{r} = \texttt{g} \\ \texttt{false} & \text{otherwise.} \end{cases}$$

That is, `f` is a polymorphic function that, when its first argument happens to be `g`, is able to test concretely for equality of its second and third arguments, even though it is given these arguments with completely abstract type.

10 Related Work

The basic metatheory of the polymorphic pi-calculus has been studied by Turner [Tur96], who also shows a strong correspondence between the polymorphic pi- and lambda-calculi by demonstrating that Milner's translations of lambda-terms into untyped pi-calculus both preserve and reflect polymorphic typing. Process calculi with weaker ML-style polymorphism have been developed by Gay [Gay93] and Vasconcelos and Honda [VH93]. A rather different style of polymorphism is considered by Liu and Walker [LW95].

Many other type systems have been proposed for process calculi. One that particularly invites comparison with the present system is the pi-calculus with input/output modalities developed

by the present authors [PS93], in which the capabilities of reading and writing on channels are distinguished and may be passed separately from one process to another. There, as in the present work, the main focus was on the effect of refined typings on behavioral equivalences; the main result was that imposing a natural directionality on the use of channels in one of Milner's encodings of the call-by-value lambda-calculus into the pi-calculus allowed us to prove that the encoding preserved beta-reduction, which was not true in the untyped case. I/O modalities, together with *variant* types, have also been used to prove the adequacy of a translation of a typed object-oriented language into π-calculus [San96].

One difference between the way capabilities are restricted by polymorphism and by I/O modalities is that, with the latter, an occurrence of a name which has lost certain capabilities can never recover them. By contrast, in the polymorphic system capabilities can increase and decrease: for instance, we may pass a value to the outside world in such a way that the receiver has no capabilities, but when the value is passed back to us it recovers its hidden capabilities.

I/O modalities can be cleanly integrated with polymorphism: for example, a variant of the polymorphic pi-calculus with I/O modalities (as well as higher-order polymorphism) forms the core of the Pict programming language [PT96].

Another class of pi-calculus type systems for which behavioral consequences have been studied are those based on linear typing [Hon93, KPT96, Hon96]. The crucial observation here is that, in the absence of global operators such as general choice, a communication occurring on a linear ("use-once") channel can never interfere with any other communication, and hence preserves the weak bisimilarity class of the process. Thus, like I/O modalities and polymorphism, linear typing not only leads to a coarser equivalence on processes (validating program transformations such as tail-call optimization), but also enables more powerful forms of algebraic reasoning about equivalence. The basic mechanisms of linearity have more recently been extended to type systems capable of guaranteeing properties such as deadlock freedom in certain cases [Yos96, Kob96].

The basic intuition behind the notion of parametricity, introduced by Strachey [Str67] and refined by Reynolds [Rey74] and others, is that a polymorphic function is parametric if its behavior is independent of (or uniform in) the type at which it is instantiated. This intuition can be phrased either intensionally — a parametric polymorphic function executes the same algorithm regardless of its type parameter — or extensionally, using Reynolds's notion of *relational parametricity* [Rey83], which expresses the uniformity of behavior of polymorphic expressions in a convenient extensional form by showing how the externally observable behaviors of different instances of a polymorphic expression are "related" in a precise way. We have adopted an intensional point of view in this paper, observing that the "abstractness" of type parameters is preserved during the evolution of a process and using this to infer behavioral properties of unknown processes; but an extensional approach would also be of interest. The main difficulty to be overcome here is that, because of the "information leakage" phenomena discussed in Section 9, it is not easy to define what it means for two instances of a process expression to be "given related inputs" or what it means for the two instances to "behave in related ways." Nor, for the same reasons, is it clear whether a more extensional account of parametricity for the pi-calculus would be much more useful than the operational one we've developed here. As far as we know, this problem has not yet been tackled satisfactorily in the lambda-calculus either, though recent work by Pitts on operational accounts of parametricity [Pit96, PS96, and unpublished notes] may be relevant.

Our proof technique based on polymorphic types yields a simple proof of equivalence between processes whose untyped behaviors have incomparable sets of traces. Proof techniques with this property are rare in the literature. Perhaps the best known is Larsen's *relativised bisimulation* [Lar87]; indeed, our method can be seen as a disciplined instance of Larsen's, in which one uses

types to express constraints on the behaviors of the observers, rather than explicitly writing all their possible behaviors.

11 Discussion

We close with a brief discussion of some additional technical issues.

Subject Reduction and Type Unification

Examples in Section 9 show that increases of type knowledge on names, that a process may achieve in an input, have to be propagated to all names of the process's type environment. This calls for type unification. A reader might wonder why it does not appear in the statement of our subject reduction theorem. A formulation of the input clause of the subject reduction theorem which makes this the increase of type information and the use of type unification explicit is the following:

11.1 Theorem [Subject reduction, more explicit input clause]: Let , and P be open. Suppose , $\vdash P$.

If $P \xrightarrow{a\{\widetilde{X}\}[\widetilde{b}]} P'$ (with $\widetilde{X}$ fresh for P and ,) then

(a) , $(a) = \updownarrow\{\widetilde{X}\}[\widetilde{U}]$, for some $\widetilde{U}$,

(b) if , $' \stackrel{\text{def}}{=}$, $\uplus\, \widetilde{b} : \widetilde{U}$ and θ the substitution s.t. , $'(\widetilde{b}) = \theta\widetilde{U}$, then , $' \vdash \theta P'$.

In clause (b), , $\uplus\, \widetilde{b} : \widetilde{U}$ denotes (first-order) type unification between , and $\widetilde{b} : \widetilde{U}$.

The two versions of subject reduction (namely the original Theorem 5.1(2) and the one above) are equivalent. clearly, the one above implies the original one; here is a proof showing the converse:

Proof: From , $\vdash P$ and Lemma 3.2, θ, $\vdash \theta P$. From $P \xrightarrow{a\{\widetilde{X}\}[\widetilde{b}]} P'$ and Lemma 4.2, $\theta P \xrightarrow{a\{\theta\widetilde{X}\}[\widetilde{b}]} \theta P'$. It also holds that (assuming that $\widetilde{Y}$ are fresh type variables)

- $(\theta,)(a) = \updownarrow\{\widetilde{Y}\}[\theta([\widetilde{Y}/\widetilde{X}]\widetilde{U})]$,
- , $'$ extends θ, ,
- , $'(\widetilde{b}) = \theta\widetilde{U} = [\theta\widetilde{X}/\widetilde{Y}](\theta([\widetilde{Y}/\widetilde{X}]\widetilde{U}))$

Therefore, from Theorem 5.1(2), we can conclude that , $' \vdash \theta P'$. □

Labeled Equivalence

In this paper we have worked in terms of barbed equivalence, where the bisimulation game between two processes is played only on internal communications and the definition itself requires congruence for parallel composition. In the ordinary *labeled* bisimulations, (like the strong and weak bisimulations of CCS [Mil89], or late and early bisimulations of the untyped pi-calculus [MPW92]) no congruence property is built into the definition, and the bisimulation game is played also on the visible actions.

The main advantage of barbed equivalence is that its definition is straightforward, even in a typed setting. On the other hand, with labeled bisimilarities, proofs for real examples require less work, because the bisimulation candidates are typically smaller (although conceptually the proofs tend to be of similar difficulty).

In the presence of polymorphism — and in general in typed calculi — finding the right definition of labeled bisimulation and proving the necessary basic properties (in particular the congruence for parallel composition) appears hard. The reason has to do with multiple "points of view" about the types of the values in a program, one of the most subtle features of polymorphism — both the universal polymorphism that we are dealing with here and the subtype polymorphism that has been considered elsewhere. When a value is transmitted abstractly from one process to another, the receiver has less information about it — and so may use fewer of its actual capabilities — than the sender. Indeed, a value may be sent with partial type information and then retransmitted under an even more abstract type, so that there may be many different points of view on the type of a single value in the same running program.

In barbed equivalence, we do not need to worry about multiple points of views. The observer is explicitly given — it is a process that runs in parallel with the tested processes — and can therefore be required to be well-typed. Then the subject-reduction theorem guarantees that it will respect the constraints on the use of channels imposed by the typing system. By contrast, in labeled bisimulation the observer is implicit — its behavior is not given beforehand — and must be typechecked *dynamically* to make sure that it behaves like a well-typed process.

Acknowledgements

We are grateful to David N. Turner for early conversations on polymorphic bisimulation; to Peter O'Hearn, Andy Pitts, and Jon Riecke, for insights about parametricity and aliasing; and to the members of the the Cambridge Interruption Club for general discussions of polymorphism in the pi-calculus. Comments from Gerard Boudol, Ole Jensen, Uwe Nestmann, Andy Pitts, Peter Sewell, Perdita Stevens, David Turner, and the anonymous referees helped us improve earlier drafts.

This work was mostly completed while Pierce was at the Computer Lab, University of Cambridge, and supported by EPSRC grant number GR/K 38403. Sangiorgi was supported by the CNET project "Modélisation de Systèmes Mobiles."

References

[Gay93] Simon J. Gay. A sort inference algorithm for the polyadic π-calculus. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[Hon93] Kohei Honda. Types for dydadic interaction. In *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523, 1993.

[Hon96] Kohei Honda. Composing processes. In Principles of Programming Languages (POPL), pages 344–357, January 1996.

[Kob96] Naoki Kobayashi. A partially deadlock-free typed process calculus. Technical report, Department of Information Science, University of Tokyo, 1996. to appear.

[KPT96] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Principles of Programming Languages*, 1996.

[Lar87] K. G. Larsen. A context dependent equivalence between processes. *Theoretical Computer Science*, 49:185–215, 1987.

[LW95] Xinxin Liu and David Walker. A polymorphic type system for the polyadic π-calculus. In *CONCUR'95: Concurrency Theory*, pages 103–116. Springer, 1995.

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

[Mil91] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer-Verlag, 1993.

[MP88] John Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[MS92] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *19th ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer Verlag, 1992.

[Pit96] A. M. Pitts. Reasoning about local variables with operationally-based logical relations. In *11th Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, Washington, 1996.

[PS93] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version to appear in *Mathematical Structures in Computer Science*, 1996.

[PS96] A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*. 1996. To appear.

[PT96] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. To appear, 1996.

[Rey74] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag LNCS 19.

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[San92] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, Department of Computer Science, University of Edinburgh, 1992.

[San96] Davide Sangiorgi. An interpretation of typed objects into typed π-calculus. To appear as Technical Report INRIA-Sophia Antipolis, 1996.

[Str67] C. Strachey. Fundamental concepts in programming languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen, August 1967.

[Tur96] David N. Turner. *The Polymorphic Pi-calulus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1996.

[VH93] Vasco T. Vasconcelos and Kohei Honda. Principal typing schemes in a polyadic pi-calculus. In *Proceedings of CONCUR '93*, July 1993. Also available as Keio University Report CS-92-004.

[Yos96] Nobuko Yoshida. Graph types for monadic mobile processes. Manuscript, May 1996.