Behavioral Equivalence in the Polymorphic Pi-Calculus*

Benjamin C. Pierce
University of Pennsylvania
Dept. of Computer & Information Science
200 South 33rd Street
Philadelphia, PA 19104-6389, USA
`bcpierce@cis.upenn.edu`

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

University of Pennsylvania Technical Report
MS-CIS-99-10

April, 1999

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

The π-calculus is at the same time a development of CCS [Mil89] and of the lambda-calculus [Chu41, Bar84]: it is, syntactically, an extension of CCS, and it is inherently "higher-order" in a sense similar to the lambda-calculus (in particular, both the lazy and the call-by-value lambda-calculus can be embedded straightforwardly in it). It has come to be regarded in the concurrency community as a paradigmatic process calculus for message-passing concurrency [MPW92, Mil91]. We study here the effect of adding polymorphic typing, in the style of the polymorphic lambda-calculus [Gir72, Rey74], to the pi-calculus.

This extension is conceptually 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 [Tur95]. 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"

*This is an expanded and revised version of a paper originally appearing in *Principles of Programming Languages*, 1997.

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 an abstract type, the usage of the value by a well-typed receiver must be independent of the actual run-time type of the value (and hence, in a suitable sense, of the value itself). 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 (carrying type T) 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." (In a full-scale programming language, $\updownarrow$ and $[\ldots]$ would be understood separately as "channel carrying" and "tuple of" type constructors, but for simplicity we roll them into a single construct here. A richer language might also include other channel type constructors, such as $\uparrow$ for output capabilities and $\downarrow$ for corresponding input capabilities [PS93].) 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

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

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

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

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 $\updownarrow[\mathit{Bool},\updownarrow[\mathit{Bool}]]$. 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

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

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. (Note that, in this example, X does not appear in the body of the input, though it is used in the typing of the body. In general, though, it may: we could, for example, pass both X and x along some other channel.) A client of *PolyId* must now send a type in addition to the other two arguments:

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

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!)

As in the polymorphic lambda-calculus, we have some freedom to choose whether the type components appearing in communications are actually passed "at run time" or are present only during typechecking. In the present theoretical study, it is helpful to actually pass types during reduction of process terms, so that we can see more clearly how the type information in a process evolves. In a practical implementation, we might prefer to erase types during compilation and execute just the underlying "bare processes"; this is the strategy adopted, for example, in the Pict compiler [PT97, Tur95].

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 $\updownarrow[Bool,\updownarrow[],\updownarrow[]]$, 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 $\updownarrow[X;X,X,\updownarrow[X,\updownarrow[],\updownarrow[]]]$:

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

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

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

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* [MS92a, San92]) and two associated proof techniques by which equivalences like the one above can be established. The basis of these proof techniques is a refinement of the usual *subject reduction* theorem. Given an open process P and an open type environment Γ 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 Γ (for instance, if a channel appears in Γ 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.

Our behavioral equivalence is contextually defined. (It is the closure of barbed bisimulation under parallel compositions.) The first proof technique directly applies the subject reduction theorem and the substitution lemma mentioned above to the definition of the behavioral equivalence. To exploit this technique on a type-closed process Q and a type-closed environment Δ (where "type-closed" means not containing free type variables) we proceed roughly thus: (1) we take an open process P and an open type environment Γ in which P is well-typed and of which Q and Δ are closed instances; (2) we use the subject reduction property to infer constraints on the behavior of P; (3) we use the substitution lemma to transfer these constraints to Q. The open P and Γ can be constructed "on the fly," while examining sequences of computation steps beginning from the initial closed 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). This technique is both sound (the equalities proved with the technique are valid) and complete (in principle, all valid process equalities can be proved with the technique). However, it requires some universal quantifications on processes. As a consequence, the set of processes that have to be exhibited in a proof is always infinite.

Our second technique for polymorphic processes is roughly a typed version of the ordinary labeled bisimilarity [Par81, Mil89, MPW92], where type environments and type substitutions are used to record both the observer's point of view on the types of the values and their actual types. (The coexistance of multiple points of view on the types of values was one of main obstacles to the formulation of proof techniques we have encountered; we postpone a detailed discussion of this point to Sections 6 and 12). Although the subject reduction and substitution lemmas do not appear in the definition of this technique, they are essential for proving its soundness. This technique is requires no universal quantification on processes. However, it is not complete: there will be some processes for which it is not applicable. When it is applicable, it yields simpler proofs. Of course, soundness without completeness is, by itself, a rather weak property (syntactic identity is also sound but not complete!). We shall argue in Section 12, by means of examples, that our second technique applies to a broad and interesting class of polymorphic processes, namely those processes that do not exhibit *information leakage* in a sense described in Section 9. More details on the comparison between the two techniques may be found in Section 12.

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. In Section 10, we indulge in a brief detour to show how information leakage can be used to implement a kind of ad-hoc overloading in ML. In Section 11, building on the examples discussed in Section 9, we show a different formulation of the subject reduction theorem, which makes explicit use of type unification; the reformulated theorem should help in understanding the use of type unification in the following sections. Sections 12 and 13 develop our second proof technique for polymorphic bisimulation and apply it to the same two examples.

Section 14 surveys related work and briefly explores 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 15 sketches how the techniques developed here might be adapted for other process equivalences, including weak bisimilarity, may- and must- testing, and the "maximum sound equivalence." Section 16 summarizes our contributions and discusses their significance.

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 $\texttt{if}\ x = y\ \texttt{then}\ P\ \texttt{else}\ Q$; and the choice construct $P + Q$, which can behave like either P or Q . The syntax of types and processes is defined as follows (with x and y ranging over channel names):

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 proposal for a programming language. As we observed in the introduction, a full-scale programming language would probably not combine the type constructors for channels, tuples, and type abstractions into a single syntactic form, but would separate them into orthogonal features (cf. [PT97]). They are unified here only 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 will write $\widetilde{x} \notin \mathrm{dom}(\Gamma)$ to mean that none of the x_i should be in the domain of the typing context Γ.

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

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 Γ if its operations respect the types declared in Γ for its free names. Formally, the *typing relation* $\Gamma \vdash P$ is the least relation closed under the following rules:

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

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

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

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

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

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

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

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

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

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

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

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

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 ill typed. This prevents a process from creating new (inert) elements of types that it knows only abstractly. (Relaxing this restriction would destroy the useful invariant that "elements" of an abstract data type can only be created in the scope where the concrete representation of those elements is known. Cf., for example, the BAD-boolean example at the end of Section 7.) 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 $\Gamma \vdash P$. This justifies "reading the rules backward" to extract typing derivations for the subexpressions of P from a typing derivation for P itself.

The typing rules admit some variations that might be useful in certain contexts. For example, we might split the rule T-OUT into two: (1) the usual output rule from the simply typed pi-calculus (in which no types are transmitted), and (2) a new rule that transmits types as well as channels, but in which the channels being sent are all considered bound by the output (as in the πI-calculus [San96]). In other words, a polymorphic output would always transmit fresh channels. This refinement would be somewhat restrictive in situations where polymorphic output is being used to achieve the effect of polymorphic function application (it would require that the arguments to a polymorphic function be "wrapped" in fresh channels), but it would make behavioral equivalence somewhat better behaved by preventing some sorts of "information leakage" due to aliasing (cf. Section 9)

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.)

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

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

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

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

Note that Γ closed and $\Gamma \vdash P$ does not imply P type-closed. For instance $\emptyset \vdash (\nu\, a{:}\updownarrow[X]\,)0$; the typing is type-closed but the process is not.

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*.

In the prefix $(\nu\widetilde{x}{:}\widetilde{S})$ of an output action we take the bindings $\widetilde{x}{:}\widetilde{S}$ to be unordered. 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. In the actions $a[\widetilde{T};\widetilde{b}]$ and $(\nu\widetilde{x}{:}\widetilde{S})\overline{a}[\widetilde{T};\widetilde{b}]$ we omit the type component when it is empty.

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

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

$$\overline{a}[\widetilde{T};\widetilde{b}].\,P \xrightarrow{\overline{a}[\widetilde{T};\widetilde{b}]} P \tag{R-Out}$$

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

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

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

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

$$\frac{!P \mid P \xrightarrow{\mu} P'}{!P \xrightarrow{\mu} P'} \tag{R-Repl}$$

$$\frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'} \tag{R-Select*}$$

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

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

Note that we have chosen to present the operational semantics in a *typed* style, since this makes it easier to see how types evolve during computation. As with polymorphic lambda-calculi, this is mainly a question of style: the type annotations in processes do not affect the applicability of any of the rules, and we could just as well erase types and execute the resulting "bare" processes under a standard pi-calculus semantics. (Actually, in recent years the lambda-calculus literature has seen a shift toward "type-passing" semantics that actually *do* make use of type information for a variety of purposes at run time.)

The transition rules do not require that either of the processed involved be well typed; we will see later, though, that if the left one is well typed, then so is the right.

Similarly, 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 closed and P' is type-closed.

4.2 Lemma: If P is type-closed and $P \xrightarrow{a[\widetilde{T};\widetilde{b}]} P'$ with $\widetilde{T}$ closed, then P' is type-closed.

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]).

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

4.4 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 R and a typing for it. In the formulation used here (Theorem 5.1), the type environment Γ 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.) Clause (1), which shows that typing is preserved by internal steps (sometimes called "reductions" in the pi-calculus literature), gives the theorem its name: it is a direct analog of the standard subject reduction property of typed lambda-calculi. The other clauses extend this property to the general case of transitions involving interactions with the environment.

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

1. If $\mu = \tau$, then $\Gamma \vdash R'$.
2. If $\mu = a[\widetilde{T}; \widetilde{b}]$, then, for some $\widetilde{X}$ and $\widetilde{U}$,
   (a) $\Gamma(a) = \updownarrow[\widetilde{X}; \widetilde{U}]$,
   (b) if Γ' extends Γ and $\Gamma'(\widetilde{b}) = \{\widetilde{T}/\widetilde{X}\}\widetilde{U}$, then $\Gamma' \vdash R'$. (Note that some of the $\widetilde{b}$ may already occur in Γ, 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) $\Gamma(a) = \updownarrow[\widetilde{X}; \widetilde{U}]$,
   (b) $(\Gamma, \widetilde{x}{:}\widetilde{S})(\widetilde{b}) = \{\widetilde{T}/\widetilde{X}\}\widetilde{U}$,
   (c) $\Gamma, \widetilde{x}{:}\widetilde{S} \vdash R'$,
   (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

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

  from which we must show

$$\begin{array}{l} \Gamma(a) = \updownarrow[\widetilde{X}; \widetilde{U}] \\ \text{if } \Gamma' \text{ extends } \Gamma \text{ and } \Gamma'(\widetilde{b}) = \{\widetilde{T}/\widetilde{X}\}\widetilde{U}, \text{ then } \Gamma' \vdash \{\widetilde{T}/\widetilde{X}\}\{\widetilde{b}/\widetilde{x}\}P. \end{array}$$

  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 $\{\widetilde{T}/\widetilde{X}\}\Gamma = \Gamma$).

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 $\Gamma \vdash P$ and $\Gamma \vdash Q$, and
  - $P \xrightarrow{(\nu\widetilde{x}:\widetilde{S})\overline{a}[\widetilde{T};\widetilde{b}]} P'$, from which the induction hypothesis gives $\Gamma, \widetilde{x}{:}\widetilde{S} \vdash P'$ (using part 3c) and $(\Gamma, \widetilde{x}{:}\widetilde{S})(\widetilde{b}) = \{\widetilde{T}/\widetilde{X}\}\widetilde{U}$ (using part 3b), and
  - $Q \xrightarrow{a[\widetilde{T};\widetilde{b}]} Q'$, from which the induction hypothesis gives $\Gamma, \widetilde{x}{:}\widetilde{S} \vdash Q'$ (using part 2b).

  Combining these with T-Com, we obtain $\Gamma, \widetilde{x}{:}\widetilde{S} \vdash P' \mid Q'$, as required.

R-PAR

- : Use a case analysis on the form of μ. One only needs the induction hypothesis and weakening (Lemma 3.1).

R-NEW

- : We are given that $R = (\nu\, x{:}S\,)P$ and $R' = (\nu\, x{:}S\,)P'$, with $\Gamma, x{:}S \vdash P$ 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 $\Gamma(a) = {\updownarrow}[\widetilde{X}; \widetilde{U}]$ and, for any Γ' extending $\Gamma, x{:}S$, that $\Gamma'(\widetilde{b}) = \{\widetilde{T}/\widetilde{X}\}\widetilde{U}$ implies $\Gamma' \vdash P'$.
    Now, suppose that Γ' extends Γ and that $\Gamma'(\widetilde{b}) = \{\widetilde{T}/\widetilde{X}\}\widetilde{U}$. Since x is a bound name, we may assume it is distinct from any name bound by Γ, so $\Gamma', x{:}S$ extends $\Gamma, x{:}S$ and we have $\Gamma', x{:}S \vdash P'$. We then obtain $\Gamma' \vdash (\nu\, x{:}S\,)P' = R'$ from T-NEW.
  - The case where μ is an output is similar.

R-OPEN

- : We are given

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

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

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

  From this, we obtain the required results as follows: For part (3a), since $x \neq a$, we have $\Gamma(a) = (\Gamma, x{:}S)(a) = {\updownarrow}[\widetilde{X}; \widetilde{U}]$. 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 $\Gamma, x{:}S$ with $\widetilde{x}{:}\widetilde{S}$ is identical to the context obtained by extending Γ 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 Γ.

5.2 Corollary: Suppose $\Gamma \vdash P$ 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) $\Gamma \vdash P'$.
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) $\Gamma(a) = {}^{\updownarrow}[\widetilde{X}; \widetilde{U}]$,
   (d) if Γ' extends Γ and $\Gamma'(\widetilde{b}) = \{\widetilde{T}/\widetilde{X}\}\widetilde{U}$, then $\Gamma' \vdash P'$.
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) $\Gamma(a) = {}^{\updownarrow}[\widetilde{X}; \widetilde{U}]$,
   (d) $(\Gamma, \widetilde{x}{:}\widetilde{S})(\widetilde{b}) = \{\widetilde{T}/\widetilde{X}\}\widetilde{U}$,
   (e) $\Gamma, \widetilde{x}{:}\widetilde{S} \vdash P'$,
   (f) each component of $\widetilde{S}$ is a channel type.

Proof: By Theorem 5.1 and Lemma 4.4. For instance, in (3), parts (a) and (b) follow from Lemma 4.4(1), parts (c) to (e) follow from Theorem 5.1, and part (f) is immediate from T-New. $\square$

6 Bisimulation

The behavioral equivalence that we use is barbed equivalence. This is an *interaction-based* bisimilarity, meaning that the bisimulation game between two processes is played only on internal communications and the definition itself requires a form of context closure. Interaction-based bisimilarities contrast with the ordinary *labeled* bisimilarities (e.g., the strong and weak bisimulations of CCS [Mil89] or the late and early bisimulations of the untyped pi-calculus [MPW92]), where 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, as well as other contextually-defined behavioral equivalence (such as testing equivalence [DH84, Hen88, BD92]) is that the definition is straightforward, even in a typed setting. On the other hand, with labeled bisimilarities, proofs for real examples require somewhat less work, because the bisimulation candidates are typically smaller. (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) can be quite 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. 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. (This difficulty can be observed both with the parametric polymorphism that we are dealing with here and with the subtype polymorphism that has been considered elsewhere [PS93].)

In formulating a barbed equivalence, on the other hand, we do not need to worry about multiple points of view. The observer is explicitly given — it is a process that runs in parallel with the tested processes — and it 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 bisimulations the observer is implicit — its behavior is not given beforehand — and it must effectively be typechecked *dynamically* to make sure that it behaves like a well-typed process. In this section, we first define barbed equivalence on polymorphic processes. We develop some useful properties, leading up to a proof technique that we show at work in Section 7 and 8. We study labeled bisimilarity for polymorphic processes in Section 12.

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 is 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) [MS92a, San92]. 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.

As usually done in testing equivalence [DH84, Hen88, BD92], barbed equivalence requires quantification on parallel compositions, not on arbitrary contexts. This is sufficient for obtaining a relation that is preserved by all constructs except input prefix (see counterexample at the end of Section 6). Requiring congruence also for input prefix would yield a strictly stronger relation (intuitively, the closure of barbed equivalence under all name substitutions), and would complicate the application of our proof techniques.

6.1 Definition: Let Δ be a closed 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 $P \stackrel{\bullet}{\sim}_\Delta Q$, 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 closed typing Γ extending Δ and for each type-closed process R such that $\Gamma \vdash R$, we have $P|R \stackrel{\bullet}{\sim}_\Gamma Q|R$.

The typing environment Δ plays no role in the definition of barbed bisimulation. It could therefore be omitted, just requiring that bisimilar processes are well typed under the some closed typing. We chose to leave the typing because it plays a useful role in our proof technique in Section 7 and 8 (and to maintain an analogy with the notation for barbed equivalence; there, the typing index is necessary, since it determines the processes that can be placed in parallel). In the remainder, we write $P \stackrel{\bullet}{\sim}_\Delta Q$ and $P \sim_\Delta Q$ without recalling that P and Q must be well-typed in Δ and that Δ is closed.

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.

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 the remainder of 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.

For the proofs, it is convenient to work with a slightly more permissive notion of bisimulation, allowing smaller relations to be given as evidence of the bisimilarity of two processes.

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

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

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

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

6.4 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].)

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

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

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

Proof: A consequence of Lemma 6.5 and of the extrusion law of Lemma 6.4. □

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

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

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

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

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

and

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

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. (For instance, if b and c are different names and $\Delta \vdash a(b).\,\texttt{if } b = c \texttt{ then } P \texttt{ else } 0$, then

$$\texttt{if } b = c \texttt{ then } P \texttt{ else } 0 \quad \sim_\Delta \quad 0$$

always holds, but

$$a(b).\,\texttt{if } b = c \texttt{ then } P \texttt{ else } 0 \quad \sim_\Delta \quad a(b).\,0$$

may not.)

7 Example: Two Boolean ADTs

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

Let $\Gamma \stackrel{\text{def}}{=} \mathit{getBools} : \updownarrow[X; X, X, \updownarrow[X, \updownarrow[], \updownarrow[]]]$. 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 closed Δ extending Γ and any closed R such that $\Delta \vdash R$. Let

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

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

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

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

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

  $$\Big(\quad (\nu\, t, f, test\,)(T_1 \mid F_1 \mid IF_1 \mid \{Bool/X\}R), \\ (\nu\, t, f, test\,)(T_2 \mid F_2 \mid IF_2 \mid \{Bool/X\}R) \quad \Big)$$

  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

  $$\Big(\quad (\nu\, t, f, test, \widetilde{p} : \updownarrow[]\,)(T_1 \mid F_1 \mid \overline{h}[c, d] \mid \{Bool/X\}R), \\ (\nu\, t, f, test, \widetilde{p} : \updownarrow[]\,)(T_2 \mid F_2 \mid \overline{h}[d, c] \mid \{Bool/X\}R) \quad \Big)$$

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

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

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

  $$\Big(\quad (\nu\, t, f, test, \widetilde{p} : \updownarrow[]\,)(N_1 \mid \overline{c}[] \mid \{Bool/X\}R), \\ (\nu\, t, f, test, \widetilde{p} : \updownarrow[]\,)(N_2 \mid \overline{c}[] \mid \{Bool/X\}R) \quad \Big)$$

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

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

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

  $$\Big(\quad (\nu\, t, f, test\,)(N1 \mid \{Bool/X\}R), \\ (\nu\, t, f, test\,)(N2 \mid \{Bool/X\}R) \quad \Big)$$

  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, say

$$(\nu\, t, f, test\,)(T_1 \mid F_1 \mid IF_1 \mid \{Bool/X\}R) \xrightarrow{\tau} A_1.$$

We reason by case analysis on the subprocess that originated the action.

1. If only the subprocess $\{Bool/X\}R$ contributes to the action, then

   $$A_1 = (\nu\, t, f, test\,)(T_1 \mid F_1 \mid IF_1 \mid R_\star)$$

   for some $R_\star$ such that $\{Bool/X\}R \xrightarrow{\tau} R_\star$. By Corollary 5.2, $R_\star = \{Bool/X\}R'$, for some R' that is well typed under the same typing as R.

   The process $(\nu\, t, f, test\,)(T_2 \mid F_2 \mid IF_2 \mid \{Bool/X\}R)$ can make a matching step thus

   $$(\nu\, t, f, test\,)(T_1 \mid F_1 \mid IF_1 \mid \{Bool/X\}R) \xrightarrow{\tau} (\nu\, t, f, test\,)(T_1 \mid F_1 \mid IF_1 \mid R_\star),$$

   and the two derivatives are again in $\mathcal{R}_2$.

2. By definition, no interaction is possible within the system $T_1 \mid F_1 \mid IF_1$.

3. The remaining case to consider is that 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 5.2(3f) 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 $\widetilde{p} \subseteq \{c, d\}$, and the input from IF_1 is

   $$IF_1 \xrightarrow{test[t,c,d]} \overline{t}[d, c].$$

Therefore we have, using the laws of Lemma 6.4,

$$A_1 \mathrel{\dot{\sim}_\Delta} (\nu\, t, f, test, \widetilde{p} : {\updownarrow}[\,]\,)(T_1 \mid F_1 \mid \bar{t}[c,d] \mid \{Bool/X\}R'). \tag{1}$$

Similarly, we infer:

$$\begin{array}{ll} & (\nu\, t, f, test\,)(T_2 \mid F_2 \mid IF_2 \mid \{Bool/X\}R) \\ \xrightarrow{\tau} \dot{\sim}_\Delta & (\nu\, t, f, test, \widetilde{p} : {\updownarrow}[\,]\,)(T_2 \mid F_2 \mid \bar{t}[d,c] \mid \{Bool/X\}R'). \end{array} \tag{2}$$

The final processes in (1) and (2) form a pair of $\mathcal{R}_3$ 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. (Note the use of $\dot{\sim}_\Delta$ in matching the transitions, which is allowed by the technique of "bisimulation up to" (Definition 6.2)).

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

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

where BAD can be any process. (The disjunction $\texttt{if}\ (b=t) \vee (b=f)\ \texttt{then}\ \bar{b}[x,y]\ \texttt{else}\ BAD$ is a syntactic sugar for $\texttt{if}\ b=t\ \texttt{then}\ \bar{b}[x,y]\ \texttt{else}\,(\texttt{if}\ b=f\ \texttt{then}\ \bar{b}[x,y]\ \texttt{else}\ BAD)$.) 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.,

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

is a trace of B_1 but not of B_2.

8 Example: Two Symbol Table Packages

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

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

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 ${\updownarrow}[\texttt{Int}]$.

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

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

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

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

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 $ST_1 \mid R \stackrel{\bullet}{\sim}_\Delta ST_2 \mid R$ for all Δ extending the type environment Γ and all R such that $\Delta \vdash R$. We define a barbed Δ-bisimulation $\mathcal{R}$ up to $\stackrel{\bullet}{\sim}_\Delta$ 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

$$\begin{pmatrix} (\nu\, \mathit{ins}, \mathit{eq}, u_1, \ldots, u_m) \\ \quad (\mathit{Loop}_1 \mid \prod_{j=1}^{m} !\overline{u_j}[t_j] \mid \{S/X\}R), \\ (\nu\, \mathit{ins}, \mathit{eq}, u_1, \ldots, u_m) \\ \quad (\mathit{Loop}_2\langle B, n\rangle \mid \prod_{j=1}^{m} !\overline{u_j}[n_j] \mid \{T/X\}R) \end{pmatrix}$$

  where

$$B \stackrel{\text{def}}{=} \{(s_i, i) \ : \ 0 \leq i < n\} \tag{3}$$

  subject to the conditions

$$\Delta, \mathit{ins} : \updownarrow[\mathtt{String}, \updownarrow[X]], \mathit{eq} : \updownarrow[X, X, \updownarrow[], \updownarrow[]], u_1 : X, \ldots, u_m : X \vdash R \tag{4}$$

$$m, n \geq 0 \tag{5}$$

$$\{s_i \ : \ 0 \leq i < n\} = \{t_j \ : \ 1 \leq j \leq m\} \tag{6}$$

$$\text{for all } 1 \leq j \leq m \text{ it holds that } 0 \leq n_j < n \tag{7}$$

$$\text{for all } 1 \leq j_1, j_2 \leq m \text{ it holds that } t_{j_1} = t_{j_2} \text{ iff } n_{j_1} = n_{j_2} \tag{8}$$

  (Condition (6) ensures that the sets of strings which have been inserted into the two tables are the same. Condition 7 ensures that the integer keys used in the second table do not exceed the parameter n of $\mathit{Loop}_2\langle B, n\rangle$. Condition 8 ensures that the two tables agree on the equalities of keys stored in corresponding positions.)

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

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

for

$$
\begin{array}{lll}
P_1 & \stackrel{\text{def}}{=} & u_{j_1}(h_1).\, u_{j_2}(h_2). \\
 & & \quad \texttt{if } h_1 = h_2 \\
 & & \quad \texttt{then } \overline{f}[].\, \mathit{Loop}_1 \\
 & & \quad \texttt{else } \overline{g}[].\, \mathit{Loop}_1 \\
P_2 & \stackrel{\text{def}}{=} & u_{j_1}(h_1).\, u_{j_2}(h_2). \\
 & & \quad \texttt{if } h_1 = h_2 \\
 & & \quad \texttt{then } \overline{f}[].\, \mathit{Loop}_2\langle B, n\rangle \\
 & & \quad \texttt{else } \overline{g}[].\, \mathit{Loop}_2\langle B, n\rangle
\end{array}
$$

with B defined as in (3) and subject to the conditions (5)-(8) plus

$$
\begin{array}{l}
\Delta, \mathit{ins} : \updownarrow[\texttt{String}, \updownarrow[X]], \mathit{eq} : \updownarrow[X, X, \updownarrow[], \updownarrow[]], \\
u_1 : X, \ldots, u_m : X, \widetilde{p} : \updownarrow[] \vdash R
\end{array} \tag{9}
$$

and

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

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

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

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

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

- $\mathcal{R}_6$ 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

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

with B defined as in (3) and subject to conditions (5)-(8) plus

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

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 6.4.

$\mathcal{R}_1$: 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 (4) 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. By Lemma 4.4, an action from $\{S/X\}R$ is a substitution instance of one from R. From (4) 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

$$\begin{array}{l}\Delta, ins : \updownarrow[\texttt{String}, \updownarrow[X]], eq : \updownarrow[X, X, \updownarrow[], \updownarrow[]],\\ \quad u_1 : X, \ldots, u_m : X, \widetilde{p} : \updownarrow[] \vdash R'\\ 1 \leq j_1, j_2 \leq m\\ \widetilde{p} \subseteq \{f, g\}\end{array} \tag{10}$$

or

$$R \xrightarrow{(\nu\widetilde{p}:\updownarrow[X])\, \overline{ins}\, [t, r]} R' \tag{11}$$

with

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

In the former case, the output from $\{S/X\}R$ is

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

and this interacts with the input

$$Loop_1 \xrightarrow{eq[u_{j_1}, u_{j_2}, f, g]} P_1$$

where P_1 is the process in the definition of $\mathcal{R}_3$. The interaction between $\{S/X\}R$ and $Loop_1$ determines an interaction in the process in the first component of a pair of $\mathcal{R}_2$:

$$\begin{aligned}(\nu\, ins, eq, u_1, \ldots, u_m)(Loop_1 \mid \textstyle\prod_{j=1}^{m} !\overline{u_j}[t_j] \mid \{S/X\}R) &\xrightarrow{\tau} \\ (\nu\, ins, eq, u_1, \ldots, u_m, \widetilde{p})(P_1 \mid \textstyle\prod_{j=1}^{m} !\overline{u_j}[t_j] \mid \{S/X\}R').\end{aligned} \tag{12}$$

Similarly, $\{T/X\}R$ and $Loop_2\langle B, n\rangle$ determine an interaction in the second component of the pair of $\mathcal{R}_2$:

$$\begin{aligned}(\nu\, ins, eq, u_1, \ldots, u_m)(Loop_2\langle B, n\rangle \mid \textstyle\prod_{j=1}^{m} !\overline{u_j}[n_j] \mid \{T/X\}R) &\xrightarrow{\tau} \\ (\nu\, ins, eq, u_1, \ldots, u_m, \widetilde{p})(P_2 \mid \textstyle\prod_{j=1}^{m} !\overline{u_j}[t_j] \mid \{T/X\}R').\end{aligned} \tag{13}$$

The derivatives of (12) and (13) form a pair of processes in $\mathcal{R}_3$.

In the case of (11), reasoning as above one shows that 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 (9) 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 (8), $n_{j_1} = n_{j_2}$ 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 $0 \leq i < n$ 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 in the previous paragraph, for instance, suppose that, in addition to x, there is a global channel g of type $\updownarrow[\updownarrow[]]$, and consider the receiver process

$$P \stackrel{\text{def}}{=} x(X; m, n, o).\,(\overline{o}[m].\,\overline{o}[n] \mid g(i).\,g(j).\,(\overline{i}[] \mid j().\,\overline{equal}[])). \tag{14}$$

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 can use a value in a way that is not warranted by the type with which the value was 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:

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

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 as 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 (14) it is o, not m or n, that is aliased. Moreover, in (14) 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. For these reasons, adopting a syntactic restriction in the calculus whereby all names that are emitted with a partially or completely abstract type are fresh (but still allowing names emitted with a concrete type to be non-fresh) does not eliminate information leakage. The only syntactic condition that we see that would remove any aliasing, and hence also any information leakage, is to require *every* name sent in a communication to be fresh, essentially restricting ourselves to the pi-I-calculus [San96]).

10 Information Leakage and Overloading

Information leakage is not peculiar to the pi-calculus: similar examples can be constructed in any setting with both polymorphism and aliasing. This section takes a brief detour from the main thread of the paper to examine some consequences of information leakage in ML. (It can be skipped at will.)

First, for a simple example similar to (14) 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. (The information that is leaking here is more than just the fact that the second and third arguments have type `int`: the reference cell provides a "covert channel" by means of which the `f` can actually examine the values of its second and third arguments, despite the fact that they are given to it abstractly.)

We now present a more substantial (possibly even useful) example. The key observation, due to Perdita Stevens, is that the simple program above can be generalized to define functions that have special behavior not just on one type, but on an arbitrary collection of types.

We begin by defining a datatype of *type descriptors*:

```
datatype 'a td = TD of 'a ref * 'a * 'a
```

Intuitively, an element of `'a td` is a runtime data structure identifying the type `'a`, in the sense that it can be used to select the "`'a branch`" of an overloaded function. Concretely, an `'a td` consists of a reference cell, which we can think of as a kind of comparison register, plus two different values of type `'a`. (It is important that they be *different* values.) Here are some typical type descriptors:

```
val intTd = TD(ref(0), 0, 1)
val boolTd = TD(ref(true), false, true)
val stringTd = TD(ref(""), "", "a")
val intListTd = TD(ref([0]), [0], [1])
```

An overloaded function with result type t will be represented as an ordinary ML function with the polymorphic type `'a td -> 'a -> t`. For example, an overloaded printing function `pr` might have type `'a td -> 'a -> string`. To apply it to a concrete argument, say the integer `5`, we first apply it to the type descriptor `intTd` to select its integer version, then to the real argument `5`.

Overloaded functions are built using following generic function, which adds a new clause to an overloaded function. Its arguments are an existing overloaded function `f` $\in$ `'a td -> 'a -> t`, a type descriptor `tdb` $|in$ `s td` for the new clause, and the new body itself, `b` $\in$ `s->t`.

```
fun addbody f tdb b =
  fn tdv => fn v =>
    let val TD(ra,a1,a2) = tdb
        val TD(rv,_,_) = tdv
    in
      ra := a1; rv := v;
      if (!ra <> a1) then b(!ra) else
        (ra := a2; rv := v;
        if (!ra <> a2) then b(!ra) else
          f tdv v)
    end
: ('a td -> 'a -> 'b) -> ''c td -> (''c -> 'b) -> 'a td -> 'a -> 'b
```

The value returned by `addbody` is an overloaded function of two arguments, `tdv` and `v`, that works by checking whether the provided type descriptor `tdv` is the same as its own type descriptor `tdb` by performing a simple experiment: set the reference cell in `tdb` to a known value `a1`, then set the reference cell in `tdv` to the "polymorphic argument" `v` and check whether the contents of `a2` have changed; if this fails, they try again using a different known value `a2` (just in case `v`=`a1`). If the two type descriptors are found to match in this way, then the value of the polymorphic parameter `v` can be extracted with concrete type and passed as parameter to the body `b`. If they do not match, then `tdv` and `v` are passed along to the original overloaded function `f` (which, assuming it was built using `addbody`, will try to match them against another type descriptor, and so on).

To verify `addbody`'s behavior, we now define the overloaded printing function `pr` described above. We begin with a dummy function that always returns the string `"Undefined"`, and build up the functionality we want by adding cases, one by one, using `addbody`. For each case, we supply a type descriptor for a type `'a` and an appropriate function mapping `'a` to `string`.

```
fun pr0 td v =
  "Undefined"
: 'a -> 'b -> string

fun pr1 td v =
  addbody pr0 intTd (fn v => "Integer:" ^ (Int.toString v)) td v
: 'a td -> 'a -> string

fun pr2 td v =
```

```
    addbody pr1 intListTd
      (fn v => "IntList:"
               ^ (List.foldr (fn (i,s) => (Int.toString i)^" "^s) "" v))
      td v
  : 'a td -> 'a -> string

  fun pr td v =
    addbody pr2 stringTd (fn v => "String:" ^ v) td v
  : 'a td -> 'a -> string
```

Now, evaluating

```
pr stringTd "hello"
```

yields `"String:hello"`, while

```
pr intListTd [1,2,3]
```

yields `"IntList:1 2 3"` and

```
pr boolTd true
```

yields `"Undefined"`, since we did not bother to give `pr` a case for booleans.

Of course, what we have done here falls short of a full ac-hoc overloading mechanism for ML. (For example, the trick only works for equality types with at least two elements.) But it gives an indication of the degree to which type information can be exposed through clever use of aliasing.

11 Subject Reduction and Type Unification

To introduce the use of type unification in the labeled bisimilarity of the next section, we present a different formulation of the input clause of the subject reduction Theorem 5.1.

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 5.1. A formulation of the input clause of the theorem which makes this increase of type information and the use of type unification explicit is below. The difference is in assertion (b): in the new assertion the initial environment Γ is refined by means of the (first-order, mixed-prefix[1]) *type unification* $\Gamma \uplus \widetilde{b}{:}\widetilde{U}$, where Γ is the initial environment, and $\widetilde{b}{:}\widetilde{U}$ shows the types with which the values $\widetilde{b}$ of the action are received by the observer. The resulting environment Γ' is the least unifier for Γ and $\widetilde{b}{:}\widetilde{U}$; that is, there is a substitution ρ such that for all $a \in \Gamma$, $(\rho\,\Gamma)(a) = \Gamma'(a)$ and, for all $b_i \in \widetilde{b}$, $(\rho\,U_i)(b_i) = \Gamma'(b_i)$; moreover, if Δ is another such unifier, then there is some substitution ρ such that $\rho\Delta = \Gamma'$. If Γ and $\widetilde{b}{:}\widetilde{U}$ have no unifiers, then $\Gamma \uplus \widetilde{b}{:}\widetilde{U}$ is undefined. First-order, mixed-prefix unification is decidable, and the least unifier is unique up to renaming of bound variables. The type unification $\Gamma \uplus \widetilde{b}{:}\widetilde{U}$ shows that the type knowledge of the final process is the unification of the previous knowledge Γ and the knowledge $\widetilde{b}{:}\widetilde{U}$ that is acquired in the input action (see also Examples 11.2 and 11.3).

[1] Mixed-prefix unification [Mil92] is a straightforward extension of first-order unification, taking into account the fact that the structures being unified may contain variable binders. It allows, for example, the types $^{\ddagger}[X; X, ^{\ddagger}[]]$ and $^{\ddagger}[Y; Y, ^{\ddagger}[]]$ to be unified, but prevents the unification of $^{\ddagger}[X; X, ^{\ddagger}[]]$ and $^{\ddagger}[X; Z, ^{\ddagger}[]]$, since the resulting substitution $\{X/Z\}$ would be ill-scoped when interpreted in the top-level environment (which has no binding for X).

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

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

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

(b) if $\Gamma' \stackrel{\text{def}}{=} \Gamma \uplus \widetilde{b}{:}\widetilde{U}$ and θ the substitution such that $\Gamma'(\widetilde{b}) = \theta\widetilde{U}$, then $\Gamma' \vdash \theta P'$.

11.2 Example: Let

$$\begin{array}{rcl} \Gamma & \stackrel{\text{def}}{=} & a{:}X, d{:}\updownarrow[X], b{:}\updownarrow[Y; \updownarrow[], \updownarrow[Y], Y] \\ P & \stackrel{\text{def}}{=} & \overline{d}[a] \mid b(Z; x, y, z).\,\overline{x}[].\,\overline{y}[z]. \end{array}$$

Then $\Gamma \vdash P$ and $P \xrightarrow{b[Y;a,c,e]} \overline{d}[a] \mid \overline{a}[].\,\overline{c}[e]$. Since $\Gamma(b) = \updownarrow[Y; \updownarrow[], \updownarrow[Y], Y]$ the new type environment for P' is least unifier for Γ and $a{:}\updownarrow[], c{:}\updownarrow[Y], e{:}Y$, namely

$$\Gamma' \stackrel{\text{def}}{=} a{:}\updownarrow[], d{:}\updownarrow[\updownarrow[]], b{:}\updownarrow[Y; \updownarrow[], \updownarrow[Y], Y], c{:}\updownarrow[Y], e{:}Y.$$

Note that Γ' has both more concrete types than Γ (for names a, d) and more type assignments (names e, c, appear in Γ' but not in Γ'). In this example, substitution θ in clause (b) of the theorem is the identity.

11.3 Example: Let

$$\begin{array}{rcl} \Gamma & \stackrel{\text{def}}{=} & a{:}\updownarrow[], b{:}\updownarrow[Y; \updownarrow[Y], Y] \\ P & \stackrel{\text{def}}{=} & b(X; y, z).\,\overline{x}[y] \mid \overline{a}[]. \end{array}$$

We have $\Gamma \vdash P$ and $P \xrightarrow{b[Y;c,a]} \overline{c}[a] \mid \overline{a}[].$. The new type environment for P' is least unifier for Γ and $a{:}Y, c{:}\updownarrow[Y]$, namely $\Gamma, c{:}\updownarrow[\updownarrow[]]$. Substitution θ in clause (b) of the theorem is $\{\updownarrow[]/Y\}$.

The two versions of subject reduction (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 $\Gamma \vdash P$ and Lemma 3.2, $\theta\Gamma \vdash \theta P$. From $P \xrightarrow{a[\widetilde{X};\widetilde{b}]} P'$ and Lemma 4.3, $\theta P \xrightarrow{a[\theta\widetilde{X};\widetilde{b}]} \theta P'$. It also holds that (assuming that $\widetilde{Y}$ are fresh type variables)

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

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

12 Labeled Bisimulation

Context-based behavioral equalities like barbed equivalence are defined in terms of a universal quantification on processes. As a consequence, the bisimulation relations used in proofs are always infinite. To avoid this problem with barbed bisimulation, we can look for *direct characterizations*, without quantification on contexts. In the case of the ordinary (untyped) barbed equivalence, such a characterization is given by *labeled bisimilarity* (in the *early* style). Roughly, P and Q are bisimilar if

$$P \xrightarrow{\mu} P' \quad \text{implies} \quad Q \xrightarrow{\mu} Q', \text{ for some } Q' \text{ bisimilar to } P' \tag{15}$$

and vice versa (on the possible transitions by Q). This is a more useful definition for proving process equalities, because it is purely coinductive and has no quantification on contexts.

In the presence of polymorphism, however, (15) is far too strong. The resulting bisimilarity coincides with untyped barbed equivalence and therefore distinguishes the boolean package implementations B_1 and B_2 of Section 7, as well as the symbol table implementations ST_1 and ST_2 of Section 8.

To define a more satisfactory labeled bisimilarity for polymorphic processes, we must deal directly with 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): namely, that polymorphism engenders multiple "points of view" about the types of the values in a program. 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 even be sent with partially abstracted 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.

Fortunately, it turns out that we do not have to deal explicitly with all these possible points of view when we define a labeled bisimilarity for polymorphic processes, but can restrict our attention to just two:

(a) the *observer's perspective*—that is, the information about the types of names that an external observer has acquired through interacting with the process;

(b) the *"God's-eye" perspective*, which records the actual types of the names that are exchanged between process and observer.

We do not need to deal explicitly with the process's own (static) view of the types of the values being exchanged with the observer or between its own subprocesses, because all these points of view have already been considered in the proof of the type soundness theorem, which guarantees that the process will only use values it is given according to the limitations of the information it has about them. The observer, on the other hand, has not been type-checked; essentially, we need to keep a record of its point of view at run time so that it can be prevented from performing any actions that would reveal it to be ill-typed.

To record the observer and "God's-eye" perspectives, we use structures that we call *closures*. These are pairs $\Delta \parallel \sigma$, where Δ is an (open) type environment, and σ is a type substitution that closes up the types in Δ. The environment Δ represents the observer's perspective, and $\sigma\Delta$ the "God's-eye" perspective. In other words, Δ collects the names known by the observer together with the types that the observer has for them, the free type variables in Δ representing type information unknown to the observer; σ shows the difference between the observer's knowledge and the real types of all names.

On closures we define a transition relation called the *allow relation*. This tells us what actions of a process a well-typed observer may test. Finally, on top of the transition relation for processes and the allow relation for closures, we define a labeled bisimilarity, called *polymorphic bisimulation*. In its definition, thanks to the use of the allow relation and by contrast with the ordinary bisimilarity of the π-calculus:

(i) Only a (possibly proper) subset of the transitions of processes are observable.

(ii) The labels of matching transitions between bisimilar processes may be syntactically different. (This is needed, for instance, to equate the symbol table implementations ST_1 and ST_2, whose immediate actions along channel *getST* have different types.)

12.1 The Allow Relation

A *typed closure* is a pair $\Delta \parallel \sigma$ where Δ is a (possibly open) type environment, and σ is a closing type substitution for Δ. The allow relation is defined as follows (the unification notation $\uplus$ in rule A-Out is explained below).

12.1.1 Definition: The *allow relation* $\Gamma \parallel \sigma \xrightarrow{\mu} \Gamma' \parallel \sigma'$, where $\Gamma \parallel \sigma$ and $\Gamma' \parallel \sigma'$ are closures, and μ a closed action, is defined by the following rules:

$$\Gamma \parallel \sigma \xrightarrow{\tau} \Gamma \parallel \sigma \qquad \text{(A-Tau)}$$

$$\frac{\Gamma(a) = {\updownarrow}[\widetilde{X};\widetilde{U}] \qquad \mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma) \cup \tilde{b} \qquad \Gamma' \text{ extends } \Gamma \qquad \Gamma'(\widetilde{b}) = \{\widetilde{T}/\widetilde{X}\}\widetilde{U}}{\Gamma \parallel \sigma \xrightarrow{a[\sigma\widetilde{T};\widetilde{b}]} \Gamma' \parallel \sigma} \qquad \text{(A-Inp)}$$

$$\frac{\widetilde{x}, \widetilde{X} \text{ fresh for } \Gamma \qquad \Gamma(a) = {\updownarrow}[\widetilde{X};\widetilde{U}] \qquad \Gamma' = \Gamma \uplus \widetilde{b}{:}\widetilde{U} \qquad \sigma'\Gamma' = \Gamma\sigma,\, \widetilde{x}{:}\widetilde{S}}{\Gamma \parallel \sigma \xrightarrow{(\nu\widetilde{x}:\widetilde{S})\overline{a}[\widetilde{T};\widetilde{b}]} \Gamma' \parallel \sigma'} \qquad \text{(A-Out)}$$

Allow-transitions have the form

$$\Gamma \parallel \sigma \xrightarrow{\mu} \Gamma' \parallel \sigma' \qquad (16)$$

and are read as follows: if an observer has knowledge Γ and σ expresses the difference between this knowledge and the real types of names, then this observer *allows* the process transition μ. The result is an observer that has knowledge Γ', and σ' is the new difference with the real types of names. In the case where μ is an input or an output action, (16) says that an interaction between the observer and the process in which μ is the action performed by the process is possible. If μ is a τ-action, then (16) always holds, with $\Gamma' = \Gamma$ and $\sigma' = \sigma$; this says that an observer always allows silent transitions of the process and these transitions do not affect the observer's knowledge.

Type unification occurs in rule A-Out for the same reason why it occurs in the second subject reduction Theorem 11.1. In A-Out, the process interacting with the observer performs an output action, hence the observer itself performs an input. From the observer's point of view, the input performed is $a[\widetilde{X};\widetilde{b}]$: thus the unification $\Gamma \uplus \widetilde{b}{:}\widetilde{U}$ in the rule, which defines the new type knowledge for the observer, is precisely that in clause (b) of Theorem 11.1. The third premise, $\sigma'\Gamma'$, shows the actual types of the names in Γ', which, since $\mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma) \cup \widetilde{x}$, is the actual type of the names in Γ plus that of the newly created names $\widetilde{x}$. If Γ and $\widetilde{b}{:}\widetilde{U}$ have no unifiers, then $\Gamma \uplus \widetilde{b}{:}\widetilde{U}$ is undefined and A-Out cannot be applied.

An example that uses A-OUT is the following: it is the analog of Example 11.2; the role of process P in Example 11.2 is played here by the observer:

$$\begin{array}{l} a : X, d : {\updownarrow}[X], b : {\updownarrow}[Y; {\updownarrow}[], {\updownarrow}[Y], Y] \parallel \{{\updownarrow}[]/X\} \xrightarrow{(\nu c:{\updownarrow}[{\updownarrow}[]], e:{\updownarrow}[])\overline{b}[{\updownarrow}[];a,c,e]} \\ a : {\updownarrow}[], d : {\updownarrow}[{\updownarrow}[]], b : {\updownarrow}[Y; {\updownarrow}[], {\updownarrow}[Y], Y], c : {\updownarrow}[Y], e : Y \parallel \{{\updownarrow}[]/Y\} \end{array}$$

Names c and e are fresh; a is received with a completely concrete type, and this determines a refinement of the types of the previously available names.

The initial environment Γ of the allow relation may also grow in rule A-INP, if the observer creates a new name and passes it to the process, as in

$$a : {\updownarrow}[{\updownarrow}[X]] \parallel \sigma \xrightarrow{a[b]} a : {\updownarrow}[{\updownarrow}[X]], b : {\updownarrow}[X] \parallel \sigma \,.$$

Below are some lemmas dealing with the allow relation that we shall need later, in the proof of the main results of the section. Lemma 12.1.2 shows that if the observer has a channel type on a name, then the observer always allows inputs of fresh values along that name, no matter what the implementation types of the action are.

12.1.2 Lemma: Suppose $\Gamma(a) = {\updownarrow}[\widetilde{X}; \widetilde{U}]$ is a closed type. Then, for all closed $\widetilde{T}$ and for all fresh $\widetilde{b}$, we have $\Gamma \parallel \sigma \xrightarrow{a[\widetilde{T};\widetilde{b}]} \Gamma, \widetilde{b}{:}\{\widetilde{T}/\widetilde{X}\}\widetilde{U} \parallel \sigma$.

Proof: Immediate from the definition of the allow relation. $\square$

The two lemmas below give some technical results about the relationship between the transition relation on processes and the allow relation on closures. Lemma 12.1.3 shows that if a well-typed process can perform an output action along a channel and the observer at least knows that the channel has a channel type, then the observer allows that action of the process. The first two premises of the lemma require that the closure agrees with the typing of the process: the process is type-closed and well-typed with respect to the actual types of the names as specified in the closure.

12.1.3 Lemma: If $\sigma\Gamma \vdash P$ and P is type-closed, $P \xrightarrow{(\nu\widetilde{x}:\widetilde{S})\overline{a}[\widetilde{T};\widetilde{b}]} P'$, and $\Gamma(a)$ is a channel type, then there are Γ' and σ' such that $\Gamma \parallel \sigma \xrightarrow{(\nu\widetilde{x}:\widetilde{S})\overline{a}[\widetilde{T};\widetilde{b}]} \Gamma' \parallel \sigma'$ with $\sigma'\Gamma' = \sigma\Gamma, \widetilde{x}{:}\widetilde{S}$.

Proof: Follows from subject reduction (Theorem 5.1), and the definitions of the allow relation and of type unification. $\square$

As explained before, an allow relation $\Gamma \parallel \sigma \xrightarrow{\mu} \Gamma' \parallel \sigma'$ represents an interaction between a process and the observer. The observer behaves as a process that is well-typed in Γ and that performs an action whose type-closed instance (according to the substitution in the closure) is the dual of μ. There should therefore be agreement between the actions performed by a process well-typed in a typing Γ and the dual of the actions allowed by an observer with the same knowledge Γ. Lemma 12.1.4(1) gives an example of this agreement, for the case where the action by the process is an output (this will be enough for our later proofs). Lemma 12.1.4(2) shows that, in an allow relation, if the action is an input and the closure does not change, then the implementation types are unique.

12.1.4 Lemma: If $\Gamma \vdash P$ and $P \xrightarrow{(\nu\widetilde{x}:\widetilde{S})\overline{a}[\widetilde{T};\widetilde{b}]} P'$, then for all σ such that σP is type-closed and $\Gamma \parallel \sigma$ is a closure, we have:

1. $\Gamma, \widetilde{x} : \widetilde{S} \parallel \sigma \xrightarrow{a[\sigma\widetilde{T};\widetilde{b}]} \Gamma, \widetilde{x} : \widetilde{S} \parallel \sigma$.

2. Let $\Gamma(a) = {\updownarrow}[\widetilde{X};\widetilde{U}]$ where each $X \in \widetilde{X}$ occurs free in $\widetilde{U}$. If $\Gamma, \widetilde{x} : \widetilde{S} \parallel \sigma \xrightarrow{a[\widetilde{T'};\widetilde{b}]} \Gamma, \widetilde{x} : \widetilde{S} \parallel \sigma$, then $\widetilde{T'} = \sigma\widetilde{T}$.

Proof: (1) From $\Gamma \vdash P$ and Theorem 5.1(3), if $\Gamma(a) = {\updownarrow}[\widetilde{X};\widetilde{U}]$, then we infer $(\Gamma, \widetilde{x} : \widetilde{S})(\widetilde{b}) = \{\widetilde{T}/\widetilde{X}\}\widetilde{U}$. The result follows by the definition of the allow relation. Part (2) follows directly from the definition of the allow relation. □

12.2 A Labeled Bisimulation

We now introduce polymorphic bisimulation. For the reasons outlined in the introduction to this section, for each process we need to have a closure that tells us the observer and the God's eye perspectives. A polymorphic bisimulation consists of tuples of the form $(\Gamma \diamond \sigma_1 \diamond P_1 \diamond \sigma_2 \diamond P_2)$ (called *configurations*) where P_1, P_2 are the processes being compared and $\Gamma \parallel \sigma_i$ $(i = 1, 2)$ is the closure relative to P_i. Intuitively, Γ represents the observer; the same Γ appears in the two closures because the two processes must be tested by the *same* observer. In the bisimilarity clauses, only the transitions of a process that are allowed by its closure are taken into account. In the labels of two matching transitions, the names must be the same, but the types may be different.

12.2.1 Definition: A *configuration* is a 5-tuple $(\Gamma \diamond P_1 \diamond \sigma_1 \diamond P_2 \diamond \sigma_2)$ such that

1. σ_1 and σ_2 are closed substitutions on $TVars(\Gamma)$;

2. P_i is type-closed and $\sigma_i\Gamma \vdash P_i$, for $i = 1, 2$.

12.2.2 Definition: A set $\mathcal{R}$ of configurations is a *polymorphic bisimulation* if, for each $(\Gamma \diamond P_1 \diamond \sigma_1 \diamond P_2 \diamond \sigma_2) \in \mathcal{R}$,

1. if $P_1 \xrightarrow{\tau} P'_1$, then there is some P'_2 such that
   (a) $P_2 \xrightarrow{\tau} P'_2$,
   (b) $(\Gamma \diamond P'_1 \diamond \sigma_1 \diamond P'_2 \diamond \sigma_2) \in \mathcal{R}$;

2. if $\Gamma \parallel \sigma_1 \xrightarrow{a[\widetilde{T_1};\widetilde{b}]} \Gamma' \parallel \sigma_1$ and $P_1 \xrightarrow{a[\widetilde{T_1};\widetilde{b}]} P'_1$, then there are P'_2 and $\widetilde{T_2}$ such that
   (a) $\Gamma \parallel \sigma_2 \xrightarrow{a[\widetilde{T_2};\widetilde{b}]} \Gamma' \parallel \sigma_2$ and $P_2 \xrightarrow{a[\widetilde{T_2};\widetilde{b}]} P'_2$,
   (b) $(\Gamma' \diamond P'_1 \diamond \sigma_1 \diamond P'_2 \diamond \sigma_2) \in \mathcal{R}$;

3. if $\Gamma \parallel \sigma_1 \xrightarrow{(\nu\widetilde{x}:\widetilde{S}_1)\overline{a}[\widetilde{T}_1;\widetilde{b}]} \Gamma' \parallel \sigma'_1$ and $P_1 \xrightarrow{(\nu\widetilde{x}:\widetilde{S}_1)\overline{a}[\widetilde{T}_1;\widetilde{b}]} P'_1$, then
   (a) Γ' is an extension of Γ (see discussion below),

   and there are P'_2, σ'_2, $\widetilde{S_2}$, and $\widetilde{T_2}$ such that

   (b) $\Gamma \parallel \sigma_2 \xrightarrow{(\nu\widetilde{x}:\widetilde{S}_2)\overline{a}[\widetilde{T}_2;\widetilde{b}]} \Gamma' \parallel \sigma'_2$ and $P_2 \xrightarrow{(\nu\widetilde{x}:\widetilde{S}_2)\overline{a}[\widetilde{T}_2;\widetilde{b}]} P'_2$,
   (c) $(\Gamma' \diamond P'_1 \diamond \sigma'_1 \diamond P'_2 \diamond \sigma'_2) \in \mathcal{R}$;

and the converse of (1-3), on the actions from P_2.

We write $P_1 \simeq_{\Gamma\|(\sigma_1,\sigma_2)} P_2$ when $(\Gamma \diamond P_1 \diamond \sigma_1 \diamond P_2 \diamond \sigma_2) \in \mathcal{R}$ for some polymorphic bisimulation $\mathcal{R}$. When σ_1 and σ_2 are both the identity substitution, we shorten this to $P_1 \simeq_\Gamma P_2$.

It is worth pausing to discuss clause (3a) of Definition 12.2.2. This clause says that type unifications required in the bisimulation, and caused by rule A-Out of the allow relation, are straightforward: the environment may be extended but not refined. Semantically, this means that the processes in the bisimulation have no information leakage (cf. Section 9). As a consequence, however, polymorphic bisimulation is not complete for barbed equivalence; that is, there are type environments and processes that are in the barbed equivalence relation but not in a polymorphic bisimulation relation. As an example, take the processes

$$
\begin{aligned}
P &\stackrel{\text{def}}{=} (\nu\, c : \updownarrow[T])(\overline{a}[\updownarrow[T]; c].\, \overline{b}[T; c].\, c(x).\, 0)\\
Q &\stackrel{\text{def}}{=} (\nu\, c : \updownarrow[T])(\overline{a}[\updownarrow[T]; c].\, \overline{b}[T; c].\, 0).
\end{aligned}
$$

If $\Gamma \stackrel{\text{def}}{=} a{:}\updownarrow[X; X], b{:}\updownarrow[Y; \updownarrow[Y]]$, then $P \sim_\Gamma Q$ A process interacting with P or Q receives c, but receives neither a value that can be sent along c nor sufficient type information to create such a value on its own (cf. T-New). As a consequence, the input $c(x)$ of P can never be consumed. However, $P \simeq_\Gamma Q$ does not hold, because P and Q may emit name c twice, first along a and then along b, and the second emission gives the observer more information on the type of c. The output along b requires, in the allow relation, a unification that is more than environment extension. We have adopted this restriction because

1. The present definition is, mathematically, much simpler to handle. We conjecture that the relation obtained by removing clause (3a) is both sound and complete for barbed equivalence, but we have not been able to prove either.

2. In the discussion of information leakage in Section 9, we advocated a programming style that avoids the leakage by protecting values emitted along polymorphic channels. On programs that follow this programming style (like those in Sections 7 and 8), the present Definition 12.2.2 is sufficient.

We leave for future work the formalization of a statement about the soundness and completeness of Definition 12.2.2 for a restricted class of processes. The technique might be complete for the processes without information leakage but, as discussed in Section 9, we do not know how to syntactically characterize this class. We also leave for future work the question of soundness and completeness of Definition 12.2.2 without clause (3a) for the whole class of processes.

In the remainder of this section we show: 1) an alternative and more compact formulation of Definition 12.2.2; 2) some "up-to" techniques for polymorphic bisimulation; 3) the soundness of polymorphic bisimulation with respect to barbed equivalence.

Before this, we present a lemma that we will find useful later. It allows us to weaken the environment of a bisimilarity, and is easily proved from the definition of bisimulation.

12.2.3 Lemma: If $P_1 \simeq_{\Gamma\|(\sigma_1,\sigma_2)} P_2$ and Γ' is an extension of Γ such that $\sigma_1\Gamma'$ and $\sigma_2\Gamma'$ are closed, then $P_1 \simeq_{\Gamma'\|(\sigma_1,\sigma_2)} P_2$.

12.3 Alternative Presentation of Bisimulation

Definition 12.2.2 separates the clauses for input, output, and τ-actions, making their different requirements on actions explicit. Below is an alternative, more compact, presentation. The *skeleton* of an action μ is what is left of μ after stripping off all types.

12.3.1 Lemma: A set $\mathcal{R}$ of configurations is a polymorphic bisimulation iff for each $(\Gamma \diamond P_1 \diamond \sigma_1 \diamond P_2 \diamond \sigma_2) \in \mathcal{R}$,

$$\Gamma \parallel \sigma_1 \xrightarrow{\mu_1} \Gamma' \parallel \sigma_1' \quad \text{and} \quad P_1 \xrightarrow{\mu_1} P_1'$$

implies

1. Γ' is an extension of Γ

and there are μ_2, σ_2', and P_2' such that

2. $\Gamma \parallel \sigma_2 \xrightarrow{\mu_2} \Gamma' \parallel \sigma_2' \quad \text{and} \quad P_2 \xrightarrow{\mu_2} P_2'$,
3. the skeleton of μ_1 is the same as the skeleton of μ_2,
4. $(\Gamma \diamond P_1' \diamond \sigma_1' \diamond P_2' \diamond \sigma_2') \in \mathcal{R}$,

and conversely for the actions of P_2.

Proof: Straightforward, from the definitions of polymorphic bisimulation and the allow relation. □

12.4 Techniques of "Bisimulation Up–to"

A labeled bisimilarity like polymorphic bisimulation can be made even more powerful by enhancing it with *up–to techniques* [Mil89, San95]. Up–to techniques are useful for reducing both the size and the number of the configurations to consider when proving bisimilarities, as one is allowed to match the transitions of the two processes being compared up to some transformations on their derivatives. We briefly sketch two simple but useful up–to techniques: *up–to weakening* and *up–to injective substitutions*. The first allows us to discard entries of a type environment; the second allows us to identify configurations that only differ by an injective substitution on names.

A *polymorphic bisimulation up to injective substitutions* is defined like polymorphic bisimulation, but with clauses (1b), (2b) and (3c), which are of the form

$$(\Gamma' \diamond P_1' \diamond \sigma_1' \diamond P_2' \diamond \sigma_2') \in \mathcal{R}, \tag{17}$$

replaced by the (weaker) clause

> there is an injective substitution on names ρ such that $(\rho\Gamma' \diamond \rho P_1' \diamond \rho\sigma_1' \diamond \rho P_2' \diamond \rho\sigma_2') \in \mathcal{R}$.

In a *polymorphic bisimulation up to weakening*, (17) is replaced by

> there are Γ' and $\widetilde{x}, \widetilde{T}$ with $\Gamma = \Gamma', \widetilde{x}{:}\widetilde{T}$ and $\widetilde{x} \cap fn(P_1', P_2') = \emptyset$ such that $(\Gamma' \diamond P_1' \diamond \sigma_1' \diamond P_2' \diamond \sigma_2') \in \mathcal{R}$.

The two techniques can be used in combination. In a *polymorphic bisimulation up to injective substitutions and up to weakening* we can discard environment entries, and we can apply an injective substitution onto the resulting configuration.

All these techniques are valid in the sense that any such "up–to bisimulation" is contained in a polymorphic bisimulation. We believe that other forms of up–to techniques known for untyped bisimilarity, such as bisimulation up to bisimilarity [Mil89] or bisimulation up to context [San95], can be adapted to polymorphic bisimulation.

12.5 Soundness of Labeled Bisimilarity

We now address the task of proving the soundness of labeled bisimilarity with respect to barbed equivalence. Most of the section is taken up by the following technical lemma, from which the theorem itself follows easily.

12.5.1 Lemma: If $P_1 \simeq_\Delta P_2$ and $\Delta \vdash R$, with R type-closed, then $P_1 \mid R \stackrel{\bullet}{\sim}_\Delta P_2 \mid R$.

Proof: Define a relation $\mathcal{R}$ as follows:

$$\mathcal{R} \stackrel{\text{def}}{=} \Big\{ (Q_1, Q_2) \;:\; \begin{array}{l} \text{there are } P_1, P_2, R, \widetilde{z}, \widetilde{Z}, \sigma_1, \sigma_2 \text{ such that (for } i = 1,2) \\ Q_i = (\nu\, \widetilde{z} : \sigma_i \widetilde{Z}\,)(P_i \mid \sigma_i R), \\ \sigma_i R \text{ is type-closed,} \\ \Delta, \widetilde{z} : \widetilde{Z} \vdash R, \text{ and} \\ P_1 \simeq_{\Delta, \widetilde{z}:\widetilde{Z} \| (\sigma_1, \sigma_2)} P_2 \Big\} \end{array}$$

We show that $\mathcal{R}$ is a barbed Δ-bisimulation. This immediately proves the lemma (taking $\widetilde{z} = \widetilde{Z} = \emptyset$ and σ_i equal to the identity substitution).

Following the definition of barbed bisimulation, we show that

1. if $(\nu\, \widetilde{z} : \sigma_1 \widetilde{Z}\,)(P_1 \mid \sigma_1 R) \downarrow_a$ then also $(\nu\, \widetilde{z} : \sigma_2 \widetilde{Z}\,)(P_2 \mid \sigma_2 R) \downarrow_a$;
2. if $(\nu\, \widetilde{z} : \sigma_1 \widetilde{Z}\,)(P_1 \mid \sigma_1 R) \xrightarrow{\tau} A_1$, then there is A_2 such that $(\nu\, \widetilde{z} : \sigma_2 \widetilde{Z}\,)(P_2 \mid \sigma_2 R) \xrightarrow{\tau} A_2$ and $(A_1, A_2) \in \mathcal{R}$.

(The converse of these two clauses can be proved in the same way.)

Suppose $(\nu\, \widetilde{z} : \sigma_1 \widetilde{Z}\,)(P_1 \mid \sigma_1 R) \downarrow_a$. This is either due to $P_1 \downarrow_a$, or to $\sigma_1 R \downarrow_a$; moreover, $a \notin \widetilde{z}$. If $\sigma_1 R \downarrow_a$ holds, then using Lemma 4.3 and 4.4 we can also infer $\sigma_2 R \downarrow_a$; hence $(\nu\, \widetilde{z} : \sigma_2 \widetilde{Z}\,)(P_2 \mid \sigma_2 R) \downarrow_a$.

Suppose now $P_1 \downarrow_a$. If this comes from an output action of P_1, then we can infer $P_2 \downarrow_a$ from Lemma 12.1.3 and the definition of bisimulation. (When applying the lemma, note that since $a \notin \widetilde{z}$, we have $a \in \mathrm{dom}(\Delta)$; since Δ is closed, $\Delta(a)$ is a channel type.) The case where $P_1 \downarrow_a$ comes from an intput action of P_1 is similar, using Lemma 12.1.2 in place of Lemma 12.1.3. (Note that if P_1 can perform an input at a, then P_1 can also perform an input at a in which all channels received are fresh.)

We now show that $(\nu\, \widetilde{z} : \sigma_1 \widetilde{Z}\,)(P_1 \mid \sigma_1 R)$ and $(\nu\, \widetilde{z} : \sigma_1 \widetilde{Z}\,)(P_2 \mid \sigma_2 R)$ can match each other's τ-transitions. The move $(\nu\, \widetilde{z} : \sigma_1 \widetilde{Z}\,)(P_1 \mid \sigma_1 R) \xrightarrow{\tau} A_1$ can come from a move of P_1 alone, from a move of $\sigma_1 R$ alone, from an interaction between P_1 and $\sigma_1 R$ where P_1 performs the input, or from an interaction between P_1 and $\sigma_1 R$ where $\sigma_1 R$ performs the input. The first two cases are simple; we only show the details of the last two cases. Below, i ranges over $\{1, 2\}$, and we abbreviate $\Delta, \widetilde{z} : \widetilde{Z}$ as Γ. We say that a variable X is *neutral* for a substitution σ if: (1) either σ is not defined on X or $\sigma(X) = X$; (2) for all $Y \neq X$ on which σ is defined it holds that $X \notin TVars(\sigma(Y))$.

We assume that if a name has a type $\updownarrow[\widetilde{X}; \widetilde{V}]$, then each $X \in \widetilde{X}$ occurs free in $\widetilde{V}$; this simplifies some steps of the proof, though the general case—where some $X \in \widetilde{X}$ may not occur free in $\widetilde{V}$—is conceptually the same.

Interaction where P_1 makes the input and $\sigma_1 R$ the output.

We have

$$P_1 \xrightarrow{a[\widetilde{T_1}; \widetilde{b}]} P_1' \qquad \sigma_1 R \xrightarrow{(\nu \widetilde{x}: \widetilde{S_1}) \overline{a}[\widetilde{T_1}; \widetilde{b}]} R_1 \tag{18}$$

and $A_1 = (\nu\,\widetilde{z} : \sigma_1\widetilde{Z}\,)(\nu\,\widetilde{x} : \widetilde{S_1}\,)(P_1' \mid R_1)$. By Lemma 4.4, there are S'', T'' and R'' such that

$$R \xrightarrow{(\nu\widetilde{x}:\widetilde{S''})\overline{a}[\widetilde{T''};\widetilde{b}]} R'' \tag{19}$$

with $\sigma_1 R'' = R_1$, $\sigma_1\widetilde{T''} = \widetilde{T_1}$, and $\sigma_1\widetilde{S''} = \widetilde{S_1}$. From (19) and Lemma 4.3,

$$\sigma_2 R \xrightarrow{(\nu\widetilde{x}:\sigma_2\widetilde{S''})\overline{a}[\sigma_2\widetilde{T''};\widetilde{b}]} \sigma_2 R''. \tag{20}$$

Moreover, by Lemma 4.1, $\sigma_i\widetilde{S''}$ are closed types. From (19) and $\Gamma \vdash R$, by Theorem 5.1(3) $\Gamma, \widetilde{x}{:}\widetilde{S''} \vdash R''$. From $\Gamma, \widetilde{x} : \widetilde{S''} \vdash R$ (which comes from $\Gamma \vdash R$ and Lemma 3.1) and (19), by Lemma 12.1.4(1)

$$\Gamma, \widetilde{x}{:}\widetilde{S''} \parallel \sigma_i \xrightarrow{a[\sigma_i\widetilde{T''};\widetilde{b}]} \Gamma, \widetilde{x}{:}\widetilde{S''} \parallel \sigma_i\ . \tag{21}$$

From $P_1 \simeq_{\Gamma\parallel(\sigma_1,\sigma_2)} P_2$ and Lemma 12.2.3,

$$P_1 \simeq_{\Gamma,\widetilde{x}:\widetilde{S''}\parallel(\sigma_1,\sigma_2)} P_2\ . \tag{22}$$

From (22), (21), $\sigma_1\widetilde{T''} = \widetilde{T_1}$, $P_1 \xrightarrow{a[\widetilde{T_1};\widetilde{b}]} P_1'$ and the definition of polymorphic bisimilar we infer that there are $\widetilde{T_2}$ and P_2' such that

$$\Gamma, \widetilde{x}{:}\widetilde{S''} \parallel \sigma_2 \xrightarrow{a[\widetilde{T_2};\widetilde{b}]} \Gamma, \widetilde{x}{:}\widetilde{S''} \parallel \sigma_2$$

and

$$P_2 \xrightarrow{a[\widetilde{T_2};\widetilde{b}]} P_2' \qquad \text{and} \qquad P_2' \simeq_{\Gamma,\widetilde{x}:\widetilde{S''}\parallel(\sigma_2,\sigma_1)} P_1'. \tag{23}$$

By Lemma 12.1.4(2), $\widetilde{T_2} = \sigma_2\widetilde{T''}$. From this, (23), and (20), we infer

$$(\nu\,\widetilde{z} : \sigma_2\widetilde{Z}\,)(P_2 \mid \sigma_2 R) \xrightarrow{\tau} (\nu\,\widetilde{z} : \sigma_2\widetilde{Z}\,)(\nu\,\widetilde{x}{:}\sigma_2\widetilde{S''}\,)(P_2' \mid \sigma_2 R'').$$

Summarizing, we have proved that:

- $(\nu\,\widetilde{z} : \sigma_i\widetilde{Z}\,)(P_i \mid \sigma_i R) \xrightarrow{\tau} (\nu\,\widetilde{z} : \sigma_i\widetilde{Z}\,)(\nu\,\widetilde{x}{:}\sigma_i\widetilde{S''}\,)(P_i' \mid \sigma_i R'')$;
- $\Gamma, \widetilde{x}{:}\widetilde{S''} \vdash R''$;
- $P_1' \simeq_{\Gamma,\widetilde{x}:\widetilde{S''}\parallel(\sigma_1,\sigma_2)} P_2'$.

This shows that the pair

$$\Big((\nu\,\widetilde{z} : \sigma_1\widetilde{Z}\,)(\nu\,\widetilde{x}{:}\sigma_1\widetilde{S''}\,)(P_1' \mid \sigma_1 R''),\ (\nu\,\widetilde{z} : \sigma_2\widetilde{Z}\,)(\nu\,\widetilde{x}{:}\sigma_2\widetilde{S''}\,)(P_2' \mid \sigma_2 R'')\Big)$$

is in $\mathcal{R}$ and concludes the case.

Interaction where P_1 makes the output and $\sigma_1 R$ the input.

We have

$$P_1 \xrightarrow{(\nu\widetilde{x}:\widetilde{S_1})\overline{a}[\widetilde{T_1};\widetilde{b}]} P_1' \qquad \sigma_1 R \xrightarrow{a[\widetilde{T_1};\widetilde{b}]} R_1 \tag{24}$$

and

$$A_1 = (\nu\,\widetilde{z} : \sigma_1\widetilde{Z}\,)(\nu\,\widetilde{x} : \widetilde{S_1}\,)(P_1' \mid R_1). \tag{25}$$

By Lemma 4.4, also R can perform an input at a, hence since $\Gamma \vdash R$, the type of a in Γ is a channel type.

Since $\sigma_1\Gamma \vdash P_1$, $P_1 \xrightarrow{(\nu\widetilde{x}:\widetilde{S_1})\overline{a}[\widetilde{T_1};\widetilde{b}]} P_1'$, and the type of a in Γ is a channel type, by Lemma 12.1.3 there are Γ' and σ_1' such that

$$\Gamma \parallel \sigma_1 \xrightarrow{(\nu\widetilde{x}:\widetilde{S_1})\overline{a}[\widetilde{T_1};\widetilde{b}]} \Gamma' \parallel \sigma_1'$$

with $\sigma_1'\Gamma' = \sigma\Gamma, \widetilde{x} : \widetilde{S}$. Since $P_1 \simeq_{\Gamma\|(\sigma_1,\sigma_2)} P_2$, there are σ_2', $\widetilde{S_2}$, $\widetilde{T_2}$ and P_2' such that

$$\Gamma \parallel \sigma_2 \xrightarrow{(\nu\widetilde{x}:\widetilde{S_2})\overline{a}[\widetilde{T_2};\widetilde{b}]} \Gamma' \parallel \sigma_2' \qquad \text{with} \quad \sigma_2'\Gamma' = \sigma_2\Gamma, \widetilde{x} : \widetilde{S_2}\ ,$$

$$P_2 \xrightarrow{(\nu\widetilde{x}:\widetilde{S_2})\overline{a}[\widetilde{T_2};\widetilde{b}]} P_2' \tag{26}$$

and $P_1' \simeq_{\Gamma'\|(\sigma_1',\sigma_2')} P_2'$. By clause (3a) of the definition of polymorphic bisimulation, there are $\widetilde{S}'$ such that

$$\Gamma' = \Gamma, \widetilde{x} : \widetilde{S}'.$$

Moreover, $\sigma_i'\widetilde{Z_i} = \sigma_i\widetilde{Z_i}$,

$$\sigma_i'R = \sigma_i R, \tag{27}$$

and, by Lemma 3.1, $\Gamma' \vdash R$. Since $\sigma_i'\Gamma' = \sigma\Gamma, \widetilde{x} : \widetilde{S}_i$, it also holds that

$$\sigma_i'\widetilde{S}' = \widetilde{S}_i. \tag{28}$$

Let $\Gamma(a) = \updownarrow[\widetilde{X};\widetilde{V}]$ (we have shown above that $\Gamma(a)$ is a channel type). We may assume $\widetilde{X}$ neutral for σ_i. We have $(\sigma_i\Gamma)(a) = \updownarrow[\widetilde{X};(\sigma_i\widetilde{V})]$. Since $\sigma_i\Gamma \vdash P_i$, $P_i \xrightarrow{(\nu\widetilde{x}:\widetilde{S}_i)\overline{a}[\widetilde{T}_i;\widetilde{b}]} P_i'$, and $\sigma_i'\Gamma' = \sigma_i\Gamma, \widetilde{x} : \widetilde{S}_i$, by Theorem 5.1(3b) we infer

$$(\sigma_i'\Gamma')(\widetilde{b}) = \{\widetilde{T}_i/\widetilde{X}\}\sigma_i\widetilde{V}. \tag{29}$$

As $\Gamma'(a) = \Gamma(a) = \updownarrow[\widetilde{X};\widetilde{V}]$, from the definition of the allow relation (rule A-Out) we infer that there are $\widetilde{W}$ such that

$$\Gamma'(\widetilde{b}) = \{\widetilde{W}/\widetilde{X}\}\widetilde{V}. \tag{30}$$

From this, (29), and $\sigma'(\widetilde{V}) = \sigma(\widetilde{V})$ (and because each $X \in \widetilde{X}$ occurs free in $\widetilde{V}$),

$$\sigma_i'\widetilde{W} = \widetilde{T}_i. \tag{31}$$

Therefore, using (31) and (27), we can rewrite the transition by R in (24) as

$$\sigma_1'R \xrightarrow{a[\sigma_1'\widetilde{W};\widetilde{b}]} R_1$$

From which, applying Lemma 4.4, we infer that there is R' such that

$$R \xrightarrow{a[\widetilde{W};\widetilde{b}]} R' \tag{32}$$

with $\sigma_1' R' = R_1$ and, applying Lemma 4.3,

$$\sigma_2' R \xrightarrow{a[\sigma_2' \widetilde{W};\widetilde{b}]} \sigma_2' R'$$

which is to say, by (31) and (27),

$$\sigma_2 R \xrightarrow{a[\widetilde{T_2};\widetilde{b}]} \sigma_2' R' \ . \tag{33}$$

Further, from $\Gamma' \vdash R$, (32) and (30), using Theorem 5.1(2), we infer $\Gamma' \vdash R'$. Moreover, by Lemma 4.2, $\sigma_i' R'$ is type-closed. From (26), (28), and (33) we can now infer

$$(\nu\, \widetilde{z} : \sigma_2 \widetilde{Z}\,)(P_2 \mid \sigma_2 R) \xrightarrow{\tau} (\nu\, \widetilde{z}{:}\sigma_2' \widetilde{Z}\,)(\nu\, \widetilde{x}{:}\sigma_2' \widetilde{S'}\,)(P_2' \mid \sigma_2' R').$$

Summarizing, we have obtained that for $\Gamma' = \Gamma, \widetilde{x}{:}\widetilde{S'}$:

- $\Gamma' \vdash R'$;
- $\sigma_i' R'$ is type-closed;
- $(\nu\, \widetilde{z} : \sigma_i \widetilde{Z}\,)(P_i \mid \sigma_i R) \xrightarrow{\tau} (\nu\, \widetilde{z}{:}\sigma_i' \widetilde{Z}\,)(\nu\, \widetilde{x}{:}\sigma_i' \widetilde{S'}\,)(P_i' \mid \sigma_i' R')$;
- $P_1' \simeq_{\Gamma' \| (\sigma_1', \sigma_2')} P_2'$.

This proves that the pair

$$\Big((\nu\, \widetilde{z}{:}\sigma_1' \widetilde{Z}\,)(\nu\, \widetilde{x}{:}\sigma_1' \widetilde{S'}\,)(P_1' \mid \sigma_1' R'),\ (\nu\, \widetilde{z}{:}\, \widetilde{Z} \sigma_2')(\nu\, \widetilde{x}{:}\sigma_2' \widetilde{S'}\,)(P_2' \mid \sigma_2' R') \Big)$$

is in $\mathcal{R}$ and concludes the case. $\square$

12.5.2 Theorem [Soundness of labeled bisimilarity wrt. barbed equivalence]: If $P_1 \simeq_\Delta P_2$, then $P \sim_\Delta Q$.

Proof: If $P_1 \simeq_\Delta P_2$, then by Lemma 12.2.3, also $P_1 \simeq_\Gamma P_2$, for all extensions Γ of Δ. By Lemma 12.5.1, we infer $P \mid R \stackrel{\centerdot}{\sim}_\Gamma Q \mid R$, for all typed-closed R such that $\Gamma \vdash R$. We therefore conclude that $P \sim_\Delta Q$. $\square$

13 Applications of Labeled Bisimilarity

We illustrate the labeled bisimilarity developed in the previous section by applying it to the examples from Sections 7 and 8.

13.1 Boolean ADTs

We begin by applying the labeled bisimulation technique (and its associated up–to techniques) to derive a proof of the equality between the two implementations B_1 and B_2 of a boolean ADT package discussed in the introduction and in Section 7.

The proof in Section 7 uses 5 infinite relations $\mathcal{R}_i$. Using labeled bisimulation techniques, each relation $\mathcal{R}_i$ collapses down to a *finite* number of elements—sometimes a *singleton*, and at most 4 elements. Below are the definitions of the new $\mathcal{R}_i$. Processes B_i, T_i, F_i, IF_i are defined as in Section 7. We abbreviate $\Gamma, t{:}X, f{:}X, test{:}\updownarrow[X, \updownarrow[], \updownarrow[]]$ as Γ', and $\{Bool/X\}$ as σ. We use id for the identity substitution; we silently "garbage-collect" parallel compositions with process 0.

- $\mathcal{R}_1$ contains the tuple $(\Gamma \diamond B_1 \diamond \mathrm{id} \diamond B_2 \diamond \mathrm{id})$.
- $\mathcal{R}_2$ contains the tuple $(\Gamma' \diamond T_1 \mid F_1 \mid IF_1 \diamond \sigma \diamond T_2 \mid F_2 \mid IF_2 \diamond \sigma)$.
- $\mathcal{R}_3$ contains the 4 tuples

  $$\Big(\Gamma', c{:}\updownarrow[], d{:}\updownarrow[] \;\diamond\; T_1 \mid F_1 \mid \overline{h}[c,d] \;\diamond\; \sigma \;\diamond\; T_2 \mid F_2 \mid \overline{h}[d,c] \;\diamond\; \sigma\Big)$$
  $$\Big(\Gamma', c{:}\updownarrow[] \;\diamond\; T_1 \mid F_1 \mid \overline{h}[c,c] \;\diamond\; \sigma \;\diamond\; T_2 \mid F_2 \mid \overline{h}[c,c] \;\diamond\; \sigma\Big)$$

  for $h \in \{t, f\}$.
- $\mathcal{R}_4$ contains the 2 tuples

  $$(\Gamma', c{:}\updownarrow[] \diamond T_1 \mid \overline{c}[] \diamond \sigma \diamond T_2 \mid \overline{c}[] \diamond \sigma)$$
  $$(\Gamma', c{:}\updownarrow[] \diamond \overline{c}[] \mid F_1 \diamond \sigma \diamond \overline{c}[] \mid F_2 \diamond \sigma).$$
- $\mathcal{R}_5$ contains the 2 tuples

  $$(\Gamma' \diamond N_1 \diamond \sigma \diamond N_2 \diamond \sigma)$$

  where either $N_i = T_i$ or $N_i = F_i$ $(i \in \{1,2\})$.

We sketch the proof that the union of these $\mathcal{R}_i$ is a polymorphic bisimulation up to injective substitutions and up to weakening.

$\mathcal{R}_1$: The processes in $\mathcal{R}_1$ have only one possible transition (up to an injective substitution); namely, abbreviating $\updownarrow[Bool, \updownarrow[], \updownarrow[]]$ as T, the transitions:

$$B1 \xrightarrow{(\nu t{:}Bool,\, f{:}Bool,\, test{:}T)\overline{getBools}[Bool;t,f,test]} T_1 \mid F_1 \mid IF_1$$

$$B2 \xrightarrow{(\nu t{:}Bool,\, f{:}Bool,\, test{:}T)\overline{getBools}[Bool;t,f,test]} T_2 \mid F_2 \mid IF_2.$$

Their actions are identical, and allowed by the closure $\Gamma \parallel \mathrm{id}$, for we have:

$$\Gamma \parallel \mathrm{id} \xrightarrow{(\nu t{:}Bool,\, f{:}Bool,\, test{:}T)\overline{getBools}[Bool;t,f,test]} \Gamma' \parallel \sigma.$$

The derivative processes and closure form the tuple of $\mathcal{R}_2$.

$\mathcal{R}_2$: Process $T_1 \mid F_1 \mid IF_1$ has several possible transitions, including

$$T_1 \mid F_1 \mid IF_1 \xrightarrow{t[c,d]} \overline{c}[] \mid F_1 \mid IF_1 \qquad \text{(for all } c, d\text{)},$$

$$T_1 \mid F_1 \mid IF_1 \xrightarrow{f[c,d]} T_1 \mid \overline{d}[] \mid IF_1 \qquad \text{(for all } c, d\text{)},$$

$$T_1 \mid F_1 \mid IF_1 \xrightarrow{test[h,c,d]} T_1 \mid F_1 \mid \overline{h}[c, d] \qquad \text{(for all } h, c, d\text{)}.$$

However, only the transitions of the form

$$T_1 \mid F_1 \mid IF_1 \xrightarrow{test[t,c,d]} T_1 \mid F_1 \mid \overline{t}[c, d] \qquad \text{(for all } c, d \notin \mathrm{dom}(\Gamma')\text{)} \tag{34}$$

$$T_1 \mid F_1 \mid IF_1 \xrightarrow{test[f,c,d]} T_1 \mid F_1 \mid \overline{f}[c, d] \qquad \text{(for all } c, d \notin \mathrm{dom}(\Gamma')\text{)} \tag{35}$$

are allowed by the closure $\Gamma' \parallel \sigma$, as for all $c, d \notin \mathrm{dom}(\Gamma')$ and for $h \in \{t, f\}$ we have:

$$\Gamma' \parallel \sigma \xrightarrow{test[h,c,d]} \Gamma', c{:}^{\updownarrow}[], d{:}^{\updownarrow}[] \parallel \sigma. \tag{36}$$

Therefore, by definition of polymorphic bisimulation, it suffices to show that process $T_2 \mid F_2 \mid IF_2$ matches transitions (34) and (35). The matching transition for (34) (transition (35) can be handled similarly) is

$$T_2 \mid F_2 \mid IF_2 \xrightarrow{test[t,c,d]} T_2 \mid F_2 \mid \overline{t}[d, c]$$

as, by (36) this action is allowed by $\Gamma' \parallel \sigma$ and, up to an injective substitution on names,

$$(\Gamma', c{:}^{\updownarrow}[], d{:}^{\updownarrow}[] \diamond T_1 \mid F_1 \mid \overline{h}[c, d] \diamond \sigma \diamond T_2 \mid F_2 \mid \overline{h}[d, c] \diamond \sigma) \in \mathcal{R}_3.$$

By contrast, the process $T_2 \mid F_2 \mid IF_2$ cannot match the other transitions of $T_1 \mid F_1 \mid IF_1$, and this is the reason why the two processes are not equated by the ordinary labeled bisimilarity of the π-calculus. In the polymorphic bisimulation, these additional transitions are not taken into account because their corresponding actions are not allowed by closure $\Gamma' \parallel \sigma$.

$\mathcal{R}_3$: A process like $T_1 \mid F_1 \mid \overline{t}[c, d]$ has several possible transitions, including input transitions emanating from T_1 and F_1, the output transition

$$T_1 \mid F_1 \mid \overline{t}[c, d] \xrightarrow{\overline{t}[c,d]} T_1 \mid F_1$$

and the τ transition

$$T_1 \mid F_1 \mid \overline{t}[c, d] \xrightarrow{\tau} \overline{c}[] \mid F_1. \tag{37}$$

However, the closure $\Gamma', c{:}^{\updownarrow}[], d{:}^{\updownarrow}[] \parallel \sigma$ only allows the τ-action in (37), and this is matched by $T_1 \mid F_1 \mid \overline{t}[d, c]$ thus

$$T_2 \mid F_2 \mid \overline{t}[d, c] \xrightarrow{\tau} \overline{c}[] \mid F_2$$

as (weakening $\Gamma', c{:}^{\updownarrow}[], d{:}^{\updownarrow}[]$ to $\Gamma', c{:}^{\updownarrow}[]$) we have

$$(\Gamma', c{:}^{\updownarrow}[] \diamond \overline{c}[] \mid F_1 \diamond \sigma \diamond \overline{c}[] \mid F_2 \diamond \sigma) \in R_4.$$

The transitions from process $T_1 \mid F_1 \mid \overline{t}[c, c]$ can be handled similarly.

$\mathcal{R}_4$: Only the output transition

$$N_1 \mid \overline{c}[] \xrightarrow{\overline{c}[]} N_1$$

is allowed by the closure $\Gamma', c{:}^{\updownarrow}[] \parallel \sigma$

$$\Gamma', c{:}^{\updownarrow}[] \parallel \sigma \xrightarrow{\overline{c}[]} \Gamma', c{:}^{\updownarrow}[] \parallel \sigma'.$$

The above transition can be matched by $N_2 \mid \overline{c}[]$, by consuming the same output prefix $\overline{c}[]$. The resulting derivatives yield, up to weakening, the tuple $(\Gamma' \diamond N_1 \diamond \sigma \diamond N_2 \diamond \sigma)$, which is in R_5.

Note that in the case of $\mathcal{R}_2$, due to the up-to injective substitutions technique, we need not consider all possible choices of names c, d: one such choice suffices. (Similarly, in the case of R_1 for the choice of names t, f and *test*.) Without the up to injective substitution technique, the relations $\mathcal{R}_2$ to $\mathcal{R}_5$ would all be infinite.

13.2 Symbol Tables

The labeled bisimulation technique can also be used to simplify the relations needed for proving the behavioural equality between the two symbol table implementations of Section 8. We omit the details, since the simplifications to the relations R_i of Section 8, as well as the proof that the union of the resulting relations is a polymorphic bisimulation up to injective substitutions and weakening, are similar to those for the boolean package example. We give just the new definition of $\mathcal{R}_2$, using the same notation as in Section 8.

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

$$\begin{array}{l}(\Gamma, \mathit{ins}{:}^{\updownarrow}[\mathtt{String}, {}^{\updownarrow}[X]], \mathit{eq}{:}^{\updownarrow}[X, X, {}^{\updownarrow}[], {}^{\updownarrow}[]], u_1{:}X, \ldots, u_m{:}X \diamond \\ \mathit{Loop}_1 \mid \prod_{j=1}^{m} !\overline{u_j}[t_j] \diamond \\ \{S/X\} \diamond \\ \mathit{Loop}_2\langle B, n\rangle \mid \prod_{j=1}^{m} !\overline{u_j}[n_j] \diamond \\ \{T/X\})\end{array}$$

where

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

subject to the conditions

$$\begin{array}{l} m, n \ge 0 \\ \{s_i \;:\; 0 \le i < n\} = \{t_j \;:\; 1 \le j \le m\} \\ \text{for all } 1 \le j \le m \text{ it holds that } 0 \le n_j < n \\ \text{for all } 1 \le j_1, j_2 \le m \text{ it holds that } t_{j_1} = t_{j_2} \text{ iff } n_{j_1} = n_{j_2}. \end{array}$$

14 Related Work

In the pi-calculus types are assigned to names; this contrasts with the lambda-calculus, where types are assigned to terms and hence provide us with an abstract view of their behaviour. There is however a close correspondence between lambda-calculus and pi-calculus types. As first noted by Turner [Tur95], Milner's translations of untyped lazy and call-by-value lambda-calculi can be extended to translations of typed lambda-calculi, by defining a mapping from lambda-calculus types to pi-calculus types. The correspondence roughly goes as follows. The translation of a lambda-term is a pi-calculus process in which a special name, often called the *location*, is chosen for interactions with the external environment. For instance, the translation of a lambda-abstraction $\lambda x.M$ receives its argument along the location name. (Something similar occurs in the translation of data values, as hinted in Section 1 where name b is the location of the boolean *True* and *False*.) In the translation of typed lambda-calculi, the type of a lambda-term becomes the type of its location name. Such correspondences on types can be established when translating functional, object-oriented, imperative languages [PS93, Kob98, San98a, RS99] (in the case of imperative languages, processes play the role of commands; as the processes of the pi-calculus, so the commands of imperative languages have all the same type). [San98b] is a survey paper on the representation of functions (typed and untyped) as processes.

A process model that, like the lambda-calculus, assigns types to terms, is Abramsky and collegues's *Interaction Categories* [AGN95]. In Interaction Categories, objects are types, morphisms are processes respecting those types, and composition is process interaction. Interaction Categories have been used to give the semantics to data-flow languages such as Lustre and Signal, and to define classes of processes that are deadlock-free in a compositional way. It is not clear at present how Interaction Categories can handle process mobility and distribution.

The basic metatheory of the polymorphic pi-calculus has been studied by Turner [Tur95], 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 system is considered by Liu and Walker [LW95]; here, polymorphism arises not from type quantification, but by explicitly declaring the possible set of types of values that a given channel may carry. Typing itself does not guarantee absence of run-time errors; for this a notion of consistent type is separately introduced.

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 is on the effect of refined typings on behavioral equivalences; the main result is 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 is 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 [San98a].

One difference between the way capabilities are restricted by polymorphism and by I/O modalities is that, with the latter, capabilities that are lost can never be recovered. By contrast, in the polymorphic system the capabilities on a channel can increase and decrease: for instance, we may pass it to the outside world in such a way that the receiver has no capabilities, but when it is passed back to us it recovers its hidden capabilities (we saw this phenomenon, for example, with the channels t and f in the boolean package of Section 7).

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 [PT97].

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, Kob98].

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. (For instance, it is not clear what relational parametricity would mean in the first example of Section 9: the behavior of the function f has discontinuity points (for $r = g$ and $x = y$) and therefore f does not obviously map related arguments into related results. Yet f is still polymorphic, in Strachey's sense of "uniform behavior.") As far as we know, this problem has not yet been tackled satisfactorily in the lambda-calculus either, though recent work by Pitts [Pit96, PS96, Pit98b, Pit98a] and Lassen on operational accounts of parametricity may be relevant.

Our proof techniques based on polymorphic types yield proofs 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 *relativized 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. Types allow a more compact representation of the constraints, and hence proof techniques that—we hope—are easier to use and more amenable to computer-aided verification.

Another proof technique in which a typed equivalence permits comparisons of processes with incomparable sets of traces was developed by Yoshida [Yos96] (cf. her Proposition 5.15 and its use in section 7). That work uses a type system where types have a graph structure to prove the full abstraction of an encoding of the polyadic π-calculus into the monadic calculus. Graphs allow expressing sophisticated communication protocols among processes, but introduce some complications in the typing rules and in the type checking. As the type systems with linearity and polymorphism, so in Yoshida's system a transition of a process may cause significant modifications in the type environment for that process. But the two techniques are fundamentally similar in the sense that both encode non-trivial constraints on process behavior using types, both employ a labeled transition relation to track changes in these types, and both develop reasoning methods based on subject reduction.

15 Other Behavioral Equivalences

The behavioral equivalences that we have considered in this paper are *strong*: af transition of a process is always matched by exactly one transition by a bisimilar process. *Weak* behavioral equivalences abstract away from the number of internal interactions that the two processes may need to match each other's external actions. Weak versions of the equivalences defined in this paper can be obtained in the standard way. Let $\Longrightarrow$ be the reflexive and transitive closure of $\xrightarrow{\tau}$, let $\stackrel{\mu}{\Longrightarrow}$ be $\Longrightarrow\xrightarrow{\mu}\longrightarrow$ (the composition of the three relations), and let $\Downarrow_a$ be $\Longrightarrow\ \downarrow_a$. Then *weak barbed* Δ*-bisimulation*, written $\dot{\approx}_\Delta$, is defined by replacing, in clause 1 of Definition 6.1, the transition $Q \xrightarrow{\tau} Q'$ with $Q \Longrightarrow Q'$ and the predicate $Q \downarrow_a$ with $Q \Downarrow_a$, and symmetrically in clause 2. *Weak barbed* Δ*-equivalence*, written $\approx_\Delta$, is defined by replacing $\dot{\sim}_\Gamma$ with $\dot{\approx}_\Gamma$. *Weak polymorphic bisimulation* is defined by an analogous modification of Definition 12.2.2: in clause 1, transition $P_2 \xrightarrow{\tau} P_2'$ becomes $P_2 \Longrightarrow P_2'$; in clauses 2 and 3, $P_2 \xrightarrow{\mu} P_2'$ becomes $P_2 \stackrel{\mu}{\Longrightarrow} P_2'$. The extension to the weak case of the results in the paper, such as the congruence properties of barbed equivalence (except, as is usual for weak bisimilarities, the congruence with respect to summation) and the soundness of polymorphic bisimilarity, is straightforward.

We believe that our proof techniques can be adapted to forms of contextually defined behavioral equivalences other than barbed equivalence. Indeed, the choice of behavioral equivalence seems to be quite an orthogonal issue. We briefly discuss some other equivalences.

Honda and Yoshida's *maximum sound theory* [HY95] roughly differs from barbed equivalence in that the quantification on contexts is placed inside the definition of bisimilarity, rather than outside. We see no technical difficulty in adapting our proof techniques to this equivalence, with the exception of the weak version of polymorphic bisimulation, for whose soundness some modifications would be need (technically speaking, we should take the *dynamic* version [MS92b] of polymorphic bisimulation).

The definition of *may testing* equivalence [DH84, Hen88, BD92] is similar to that of weak barbed equivalence, but the bisimulation game on interactions is removed. In the definition of *must testing* in addition, the observability predicates $P \Downarrow_a$ indicate the guarantee, rather than the possibility, that P will react at a. To adapt our first proof technique for barbed equivalence (Sections 7 and 8) to may and must testing, we would replace the co-inductive reasoning on the possible interactions of processes by inductive reasoning on the length of sequences of interactions that lead to a state in which an observability predicate holds. Weak polymorphic bisimulation would still be a sound technique for may testing; to make it sound for must testing, divergence should be taken into account, following Walker [Wal88]. The idea of polymorphic bisimulation in Section 12 can also be integrated with the definition of trace equivalence, which would yield another proof technique for may testing without context quantification.

16 Conclusions

We have investigated the behavioral consequences of polymorphism in a concurrent setting, studying the constraints that polymorphism imposes on well-typed processes and observing some surprising and subtle issues due to the interaction between polymorphism and aliasing—both in concurrent languages such as pi-calculus and in sequential languages such as ML. We have presented two proof techniques for establishing behavioral equivalence between polymorphic processes and illustrated their applications on some small, but nontrivial, examples.

Polymorphism is popular in programming languages, but proof techniques for reasoning about polymorphic terms are rare. The best known techniques are based on Reynolds's relational parametricity. The examples and the discussions in Section 9 and 14 suggest that relational parametricity is not obviously portable onto higher-order languages with side effects, not even sequential languages such as ML. The proof techniques proposed in this paper are, to our knowledge, the first for polymorphic concurrent languages. Combined with encodings of typed lambda-calculi into the pi-calculus—extending Turner's encoding of the polymorphic lambda-calculus [Tur95]—the techniques may also be useful for reasoning about sequential languages. We are not aware of other behavioral techniques for higher-order sequential polymorphic languages with aliasing.

More generally, developing proof techniques for higher-order languages with aliasing has proved a difficult task even when the language is not polymorphic. For instance, it may be difficult to prove that a bisimilarity is a congruence relation. The pi-calculus, as a name-passing calculus, is not syntactically higher-order (the values exchanged in communications do not contain terms), but it is semantically higher-order, as witnessed by embeddings of lambda-calculi and higher-order process calculi [Mil90, San92]. As a consequence, proof techniques developed for the pi-calculus may be useful on other higher-order languages, either by directly applying the technique to the language, or indirectly, by translating the language into the pi-calculus.

The ideas presented in this paper have also inspired proof techniques for subtype polymorphism [BS98] and for cryptography [BDP99] in pi-calculus-related languages.

The proof methods we have proposed are quite manual, but may represent first steps towards more automatic, computer-aided methods. The integration of the labeled bisimilarity of Section 12 with existing tools for automatic or semi-automatic verification on pi-calculus processes, such as the *Mobility Workbench* [VM94] or Hirschkoff's environment based on the theorem prover COQ [Hir97], appears challenging but promising. By contrast, the technique for barbed equivalence in Sections 7 and 8 does not seem to fit a fully automatic tool such as the Mobility Workbench, though a theorem prover implementation like Hirschkoff's might be feasible.

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 for POPL and JACM 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; it was completed with the aid of NSF Career grant CCR-9701826, *Principled Foundations for Programming with Objects*. Sangiorgi was supported by the CNET project *Modélisation de Systèmes Mobiles*.

References

[AGN95] S. Abramsky, S. J. Gay, and R. Nagarajan. Interaction categories and the foundations of typed concurrent programming. In *Deductive Program Design: Proceedings of the 1994 Marktoberdorf Summer School*, NATO ASI Series F, 1995.

[Bar84] H. P. Barendregt. *The Lambda Calculus.* North Holland, revised edition, 1984.

[BD92] M. Boreale and R. De Nicola. Testing equivalence for mobile processes. In R. Cleaveland, editor, *CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*, pages 2–16. Springer Verlag, 1992. To appear in *Information and Computation.*

[BDP99] M. Boreale, R. De Nicola, and R. Pugliese. Proof techniques for cryptographic processes. Submitted for publication, 1999.

[BS98] M. Boreale and D. Sangiorgi. Bisimulation in name-passing calculi without matching. In *13th LICS Conf.* IEEE Computer Society Press, 1998.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[DH84] R. De Nicola and R. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[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. A summary appeared in the Proceedings of the Second Scandinavian Logic Symposium (J.E. Fenstad, editor), North-Holland, 1971 (pp. 63–92).

[Hen88] Matthew Hennessey. *Algebraic Theory of Processes.* MIT Press, 1988.

[Hir97] D. Hirschkoff. A full formalisation of π-calculus theory in the calculus of constructions. In Elsa Gunter, editor, *Proceedings of the International Conference on Theorem Proving in Higher Order Logics*, volume 1275 of *Lecture Notes in Computer Science.* Springer Verlag, 1997.

[Hon93] Kohei Honda. Types for dyadic 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.

[HY95] K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995.

[Kob98] N. Kobayashi. A partially deadlock-free typed process calculus. *Transactions on Programming Languages and Systems*, 20(2):436–482, 1998. A preliminary version in *12th Lics Conf.* IEEE Computer Society Press 128–139, 1997.

[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.

[Mil90] Robin Milner. Functions as processes. Research Report 1154, INRIA, Sofia Antipolis, 1990. Final version in *Journal of Mathematical Structures in Computer Science* 2(2):119–141, 1992.

[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. Appeared in *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.

[Mil92] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, October 1992.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proc. 12th ACM Symp. on Principles of Programming Languages,* 1985.

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

[MS92a] 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.

[MS92b] U. Montanari and V. Sassone. Dynamic congruence vs. progressing bisimulation for CCS. *Fundamenta Informaticae*, XVI(2):171–199, 1992.

[Par81] D.M. Park. Concurrency on automata and infinite sequences. In P. Deussen, editor, *Conf. on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*. Springer Verlag, 1981.

[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.

[Pit98a] A. M. Pitts. Existential types: Logical relations and operational equivalence. In *Proc. ICALP'98*, volume 1443, pages 309–326, 1998.

[Pit98b] A.M. Pitts. Parametric polymorphism and operational equivalence. volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[PS93] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science*, Vol. 6, No. 5, 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.

[PT97] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press, 1999.

[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).

[RS99] C. Rckl and D. Sangiorgi. A pi-calculus semantics of Concurrent Idealised ALGOL. To appear in Proc. Fossacs 99, Springer Verlag, 1999.

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

[San95] D. Sangiorgi. On the proof method for bisimulation. In J. Wiedermann and P. Háiek, editors, *Proc. MFCS'95*, volume 969 of *Lecture Notes in Computer Science*, pages 479–488. Springer Verlag, 1995. Full version to appear in J. Math. Structures in Comp. Sci.

[San96] D. Sangiorgi. π-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science*, 167(2):235–274, 1996.

[San98a] D. Sangiorgi. An interpretation of typed objects into typed π-calculus. *Information and Computation*, 143(1):34–73, 1998.

[San98b] D. Sangiorgi. Interpreting functions as pi-calculus processes: a tutorial. Technical Report RR-3470, INRIA-Sophia Antipolis, 1998.

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

[Tur95] David N. Turner. *The Polymorphic Pi-calulus: Theory and Implementation.* PhD thesis, University of Edinburgh, 1995.

[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.

[VM94] B. Victor and F. Moller. The Mobility Workbench — a tool for the π-calculus. In D. Dill, editor, *Proc. CAV'94*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.

[Wal88] D. J. Walker. Bisimulation and divergence in CCS. In *Proc. 3rd Symp. on Logic in Computer Science (LICS 88)*, pages 186–192. IEEE Computer Society Press, 1988.

[Yos96] N. Yoshida. Graph types for monadic mobile processes. In *Proc. FST & TCS*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–386. Springer Verlag, 1996.