Full Abstraction for Polymorphic Pi-Calculus

Alan Jeffrey[1,2,⋆] and Julian Rathke[3]

[1] Bell Labs, Lucent Technologies, Chicago, IL, USA
[2] DePaul University, Chicago, IL, USA
[3] University of Sussex, Brighton, UK

Abstract. The problem of finding a fully abstract model for the polymorphic π-calculus was stated in Pierce and Sangiorgi's work in 1997 and has remained open since then. In this paper, we show that a slight variant of their language has a direct fully abstract model, which does not depend on type unification or logical relations. This is the first fully abstract model for a polymorphic concurrent language. In addition, we discuss the relationship between our work and Pierce and Sangiorgi's, and show that their conjectured fully abstract model is, in fact, sound but not complete.

1 Introduction

Finding sound and complete models for languages with polymorphic types is notoriously difficult. Consider the following implementation of a polymorphic 'or' function in Java 5.0 [17]:

```
static<X> X or (X t, X a, X b) {
  if (a == t) { return a; } else { return b; }
}
```

This implementation of `or` takes a type parameter `X`, which will be instantiated with the representation chosen for the booleans, together with three parameters of type `X`: a constant for 'true', and the values to be 'or'ed. This function can be called in many different ways, for example[1]:

```
or.<int> (1, 0, 1); or.<bool> (true, false, true);
```

In each case, there is no way for the callee to determine the exact type the caller instantiated for `X`, and so *no matter what implementation for* `or` *is used*, there is no observable difference between the above program and the following:

```
or.<int> (1, 0, 1); or.<string> ("true", "false", "true");
```

⋆ This material is based upon work supported by the National Science Foundation under Grant No. 0430175.

[1] Java purists should note that this discussion assumes for simplicity that downcasting and reflection are not being used, and a particular implementation of autoboxing, for example the code `or.<int> (1, 0, 1)` is implemented as `Integer x = new Integer(1); Integer y = new Integer(0); or.<Integer> (x, y, x)`.

V. Sassone (Ed.): FOSSACS 2005, LNCS 3441, pp. 266–281, 2005.
© Springer-Verlag Berlin Heidelberg 2005

or the following:

```
or.<int> (1, 0, 1); or.<int> (2, 3, 2);
```

However, there *is* an observable difference between the above programs and:

```
or.<int> (1, 0, 1); or.<int> (1, 0, 1);
```

since we can use the following implementation of `or` to distinguish them:

```
static Object x=null;
static<X> X or (X t, X a, X b) {
  if (a == x) { System.out.println ("hello"); } else { x=a; }
  if (a == t) { return a; } else { return b; }
}
```

This example demonstrates some subtleties with polymorphic languages: the presence of impure features (such as mutable fields in this case) and equality testing (such as `a == x` in this case) can significantly impact the distinguishing power of tests. In the case of pure languages such as System F [10], the technique of *logical relations* [27, 24] can be used to establish equivalence of all of the above calls to `or`, which is evidently broken by the addition of impurity and equality testing.

Much of the work in finding models of pure polymorphic languages comes in finding appropriate techniques for modelling *parametricity* [26, 27] to show that programs are completely independent of the instantiations for their type parameters. Such parametricity results are surprisingly strong, and can be used to establish 'theorems for free' [31] such as the functoriality of the list type constructor. The strength of the resulting theorems, however, comes at a cost: the proof techniques required to establish them are quite difficult. In particular, even proving the existence of logical relations is problematic in the presence of recursive types [24].

In this paper, we show that providing models for impure polymorphic languages with equality testing can be surprisingly straightforward. We believe that the techniques discussed here will extend to the polymorphic features of languages such as Java 5.0 [17], and C# 2.0 [7]: F-bounded polymorphism [5], subtyping, recursive types and object features. In this paper, we will investigate a minimal impure polymorphic language with equality testing, based on Pierce and Sangiorgi's work [23] on a polymorphic extension of Milner *et al.*'s [21, 20] π-calculus.

Pierce and Sangiorgi have established a sound model for a polymorphic π-calculus, but they only conjectured completeness [23–Sec. 12.2]. In this paper, we develop a sound and complete model for a polymorphic π-calculus: the resulting model and proof techniques are remarkably straightforward. In particular, our model makes no use of type unification, which is an important feature of Pierce and Sangiorgi's model. We then compare our model to theirs, and show that ours is strictly finer: hence we have resolved their outstanding conjecture, by demonstrating their model to be sound but not complete.

This is the first sound and complete model for a polymorphic π-calculus: Pierce and Sangiorgi [23] and Honda *et al.* [3] have established soundness results, but not completeness.

$$
\begin{array}{rll}
a,b,c,d & & \text{(Names)} \\
x,y,z & & \text{(Variables)} \\
n,m & ::= a \mid x & \text{(Values)} \\
P,Q,R & ::= n(\vec{X};\vec{x}:\vec{T})\,.\,P \mid \overline{n}\langle\vec{T};\vec{n}\rangle \mid \mathbf{0} \mid P \mid Q & \text{(Processes)} \\
 & \quad \mid \nu(a:T)P \mid\, !P \mid \mathsf{if}\, n = m \,\mathsf{then}\, P \,\mathsf{else}\, Q &
\end{array}
$$

Fig. 1. Syntax

2 An Asynchronous Polymorphic Pi-Calculus

The language we investigate in this paper is an asynchronous variant of Pierce and Sangiorgi's polymorphic π-calculus. This is an extension of the π-calculus with type-passing in addition to value-passing.

2.1 Syntax

The syntax of the asynchronous polymorphic π-calculus is given in Figure 1. The syntax makes use of types (ranged over by T,U,V,W) and type variables (ranged over by X,Y,Z), which are defined in Section 2.3.

Definition 1 (Free identifiers). *Write* $\mathsf{fn}(P)$ *for the free names of P,* $\mathsf{fn}(n)$ *for the free names of n,* $\mathsf{fv}(P)$ *for the free variables of P,* $\mathsf{fv}(n)$ *for the free variables of n,* $\mathsf{ftv}(P)$ *for the free type variables of P and* $\mathsf{ftv}(T)$ *for the free type variables of T.*

Definition 2 (Substitution). *Let* σ *be a substitution of the form* $(\vec{V}/\vec{X};\vec{n}/\vec{x})$*, and let* $n[\sigma]$*,* $T[\sigma]$ *and* $P[\sigma]$ *be defined to be the capture-free substitution of type variables* $\vec{X}$ *by types* $\vec{V}$ *and variables* $\vec{x}$ *by values* $\vec{n}$*, defined in the normal fashion. Let the domain of a substitution* $\mathsf{dom}(\sigma)$ *be defined as* $\mathsf{dom}(\vec{V}/\vec{X};\vec{n}/\vec{x}) = \{\vec{X},\vec{x}\}$.

Definition 3 (Process contexts). *A process context* $C[\cdot]$ *is a process containing one occurrence of a 'hole'* $(\cdot)$*. Write* $C[P]$ *for the process given by replacing the hole by P.*

We present an example process, following [23], in the untyped π-calculus, in which we implement a boolean abstract datatype as:

$$\nu(t)\nu(f)\nu(test)(\overline{getBools}\langle t,f,test\rangle \mid !t(x,y)\,.\,\overline{x}\langle\rangle \mid !f(x,y)\,.\,\overline{y}\langle\rangle \mid !test(b,x,y)\,.\,\overline{b}\langle x,y\rangle)$$

This process generates new channels t, f and $test$, which it publishes on a public channel *getBools*. It then waits for input on channel t: when it receives a pair (x,y) of channels, it sends a signal on x. The same is true for channel f except that it sends the signal on y. Finally, on a test channel we wait to be sent a boolean b (which should either be t or f) together with a pair $(x.y)$ of channels, and just forwards the pair on to b, which chooses whether to signal x or signal y as appropriate. This can be typed as:

$$
\begin{aligned}
B_1 \stackrel{\text{def}}{=} \nu(t : Bool)\nu(f : Bool)\nu(test : Test(Bool))(\\
&\overline{getBools}\langle Bool; t, f, test\rangle \mid \\
&!t(x : Signal, y : Signal)\,.\,\overline{x}\langle\rangle \mid \\
&!f(x : Signal, y : Signal)\,.\,\overline{y}\langle\rangle \mid \\
&!test(b : Bool, x : Signal, y : Signal)\,.\,\overline{b}\langle x, y\rangle \\
)
\end{aligned}
$$

where we define:

$$Signal \stackrel{\text{def}}{=} \updownarrow[] \quad Bool \stackrel{\text{def}}{=} \updownarrow[Signal, Signal] \quad Test(T) \stackrel{\text{def}}{=} \updownarrow[T, Signal, Signal]$$

The interesting typing is for the channel *getBools* where the implementation of booleans is published:

$$getBools : \updownarrow[X; X, X, Test(X)]$$

that is, the implementation type *Bool* is never published: instead we just publish an abstract type X together with the values $t : X$, $f : X$ and $test : Test(X)$. Since the implementing type is kept abstract, we should be entitled to change the implementation without impact on the observable behaviour of the system, for example by uniformly swapping the positions of x and y:

$$
\begin{aligned}
B_2 \stackrel{\text{def}}{=} \nu(t : Bool)\nu(f : Bool)\nu(test : Test(Bool))(\\
&\overline{getBools}\langle Bool; t, f, test\rangle \mid \\
&!t(x : Signal, y : Signal)\,.\,\overline{y}\langle\rangle \mid \\
&!f(x : Signal, y : Signal)\,.\,\overline{x}\langle\rangle \mid \\
&!test(b : Bool, x : Signal, y : Signal)\,.\,\overline{b}\langle y, x\rangle \\
)
\end{aligned}
$$

As Pierce and Sangiorgi observe, as untyped processes B_1 and B_2 are easily distinguished, for example by the testing context:

$$T \stackrel{\text{def}}{=} \cdot \mid \nu(a)\nu(b)(getBools(t, f, test)\,.\,\overline{t}\langle a, b\rangle \mid a()\,.\,\overline{c}\langle\rangle \mid b()\,.\,\overline{d}\langle\rangle)$$

However, this process does not typecheck, since when we come to typecheck T, the channel t has abstract type X, not the implementation type *Bool*. We expect any sound and complete model to consider B_1 and B_2 equivalent.

An illustrative example of a contextual inequivalence is given below. For some generative type T consider the following processes:

$$
\begin{aligned}
L &= \nu(b : \updownarrow[T], c : \updownarrow[T], d : T)(\overline{a}\langle T, T; b, b, c, d\rangle \mid c(y : T)\,.\,\overline{\mathsf{fail}}\langle\rangle) \\
L' &= \nu(b : \updownarrow[T], c : \updownarrow[T], d : T)(\overline{a}\langle T, T; b, b, c, d\rangle \mid c(y : T)\,.\,\mathbf{0})
\end{aligned}
$$

and a type environment Γ which contains only $a : \updownarrow[X, Y; \updownarrow[X], \updownarrow[Y], \updownarrow[Y], X]$ and a suitable type for fail. Now it may at first appear that L and L' should be considered equivalent with respect to the type information in Γ as the private name d is only released along channel a at some abstract type represented by X, say. And the private name c is only

$$\mu ::= \tau \mid c(\vec{U};\vec{b}) \mid \nu(\vec{a}:\vec{T})\overline{c}\langle\vec{U};\vec{b}\rangle \quad \text{(Untyped Labels)}$$

$$\frac{}{c(\vec{X};\vec{x}:\vec{T}).P \xrightarrow{c(\vec{U};\vec{b})} P[\vec{U}/\vec{X};\vec{b}/\vec{x}]}\ \text{(R-In)} \qquad \frac{}{\overline{c}\langle\vec{U};\vec{b}\rangle \xrightarrow{\overline{c}\langle\vec{U};\vec{b}\rangle} \mathbf{0}}\ \text{(R-Out)}$$

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

$$\frac{P \xrightarrow{c(\vec{U};\vec{b})} P' \quad Q \xrightarrow{\nu(\vec{a}:\vec{T})\overline{c}\langle\vec{U};\vec{b}\rangle} Q' \quad \{\vec{a}\}\cap\mathsf{fn}(P)=\emptyset}{P\mid Q \xrightarrow{\tau} \nu(\vec{a}:\vec{T})(P'\mid Q')}\ \text{(R-Com)}$$

$$\frac{P \xrightarrow{\mu} P' \quad a\notin\mathsf{fn}(\mu)\cup\mathsf{bn}(\mu)}{\nu(a:T)P \xrightarrow{\mu} \nu(a:T)P'}\ \text{(R-New)} \qquad \frac{P \xrightarrow{\nu(\vec{a}:\vec{T})\overline{c}\langle\vec{U};\vec{b}\rangle} P' \quad a\in\{\vec{b}\}\setminus\{c,\vec{a}\}}{\nu(a:T)P \xrightarrow{\nu(\vec{a}:\vec{T},a:T)\overline{c}\langle\vec{U};\vec{b}\rangle} P'}\ \text{(R-Open)}$$

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

$$\frac{P \xrightarrow{\mu} P'}{\mathsf{if}\ a=a\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\mu} P'}\ \text{(R-Test-T)} \qquad \frac{a\neq b \quad Q \xrightarrow{\mu} Q'}{\mathsf{if}\ a=b\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\mu} Q'}\ \text{(R-Test-F)}$$

Fig. 2. Untyped Labelled Transitions $P \xrightarrow{\mu} P'$ (eliding symmetric rules for $P\mid Q$)

released as a channel which carries values of abstract type Y, say. In order to distinguish these processes a test term would need to obtain a value of type Y to send on c. However, there is a testing context which allows the name d to be cast to type Y:

$$R = a(X,Y;z:\updownarrow[X],z':\updownarrow[Y],z'':\updownarrow[Y],x:X).(\overline{z}\langle x\rangle \mid z'(y:Y).\overline{z''}\langle y\rangle)$$

It is easy to check that this process is well-typed with respect to Γ. Here, when R communicates with L and L', the vector of fresh names is received along a and the variables z and z' are aliased so that a further internal communication within R sends d as if it were of type X but receives it as if it were of type Y. It can then be sent along c to interact with the remainder of L and L' to distinguish them.

2.2 Dynamic Semantics

The untyped transition semantics for the asynchronous polymorphic π-calculus is given in Figure 2, and is the same as Pierce and Sangiorgi's. We define the free names of a label $\mathsf{fn}(\mu)$ as $\mathsf{fn}(\tau)=\emptyset$, $\mathsf{fn}(c(\vec{U};\vec{b}))=\{c,\vec{b}\}$ and $\mathsf{fn}(\nu(\vec{a}:\vec{T})\overline{c}\langle\vec{U};\vec{b}\rangle)=\{c,\vec{b}\}\setminus\{\vec{a}\}$. We also define the bound names of a label $\mathsf{bn}(\mu)$ as $\mathsf{bn}(\tau)=\mathsf{bn}(c(\vec{U};\vec{b}))=\emptyset$ and $\mathsf{bn}(\nu(\vec{a}:\vec{T})\overline{c}\langle\vec{U};\vec{b}\rangle)=\{\vec{a}\}$. The untyped semantics is useful for defining the run-time behaviour of processes, but is not immediately appropriate for defining a notion of equivalence, as it distinguishes terms such as B_1 and B_2 which cannot be distinguished by any

$$
\begin{array}{rl}
X,Y,Z & \text{(Type Variables)} \\
T,U,V,W ::= & X \mid \updownarrow[\vec{X};\vec{T}] \quad \text{(Types: } X \text{ is non-generative, } \updownarrow[\vec{X};\vec{T}] \text{ is generative)} \\
\Gamma,\Delta ::= & \vec{X};\vec{n}:\vec{T} \quad \text{(Typing Contexts)}
\end{array}
$$

$$\frac{X \in \Gamma}{\Gamma \vdash X}\ \text{(T-TVAR)} \qquad \frac{\vec{X},\Gamma \vdash \vec{T} \quad \{\vec{X}\} \cap \mathsf{dom}(\Gamma) = \emptyset \quad \vec{X} \text{ disjoint}}{\Gamma \vdash \updownarrow[\vec{X};\vec{T}]}\ \text{(T-CHAN)}$$

$$\frac{\vec{X} \vdash \vec{T}}{\vec{X};\vec{n}:\vec{T} \vdash \diamond}\ \text{(T-ENV)} \qquad \frac{\Gamma \vdash \diamond \quad (n:T) \in \Gamma}{\Gamma \vdash n:T}\ \text{(T-VAL)}$$

$$\frac{\Gamma \vdash n : \updownarrow[\vec{X};\vec{T}] \quad \vec{X},\Gamma,\vec{x}:\vec{T} \vdash P \quad \{\vec{X},\vec{x}\} \cap \mathsf{dom}(\Gamma) = \emptyset \quad \vec{x} \text{ disjoint}}{\Gamma \vdash n(\vec{X};\vec{x}:\vec{T}).P}\ \text{(T-IN)}$$

$$\frac{\Gamma \vdash n : \updownarrow[\vec{X};\vec{U}] \quad \Gamma \vdash \vec{n} : \vec{U}[\vec{T}/\vec{X}]}{\Gamma \vdash \overline{n}\langle\vec{T};\vec{n}\rangle}\ \text{(T-OUT)}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0}}\ \text{(T-NIL)} \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q}\ \text{(T-PAR)}$$

$$\frac{\Gamma,a:T \vdash P \quad a \notin \mathsf{dom}(\Gamma) \quad \mathsf{ftv}(T) \subseteq \mathsf{dom}(\Gamma) \quad T \text{ is generative}}{\Gamma \vdash \nu(a:T)P}\ \text{(T-NEW)}$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash\ !P}\ \text{(T-REPL)} \qquad \frac{\Gamma \vdash n:T \quad \Gamma \vdash m:U \quad \Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash \mathsf{if}\ n = m\ \mathsf{then}\ P\ \mathsf{else}\ Q}\ \text{(T-TEST-W)}$$

Fig. 3. Type System, with judgements $\Gamma \vdash T$, $\Gamma \vdash \diamond$, $\Gamma \vdash n:T$ and $\Gamma \vdash P$

well-typed environment:

$$
\begin{array}{ll}
B_1 & \xrightarrow{\nu(t:Bool,f:Bool,test:Test(Bool))\overline{getBools}\langle Bool;t,f,test\rangle} \xrightarrow{t(a,b)} \xrightarrow{\overline{a}\langle\rangle} \\
B_2 & \xrightarrow{\nu(t:Bool,f:Bool,test:Test(Bool))\overline{getBools}\langle Bool;t,f,test\rangle} \xrightarrow{t(a,b)} \xrightarrow{\overline{b}\langle\rangle}
\end{array}
$$

These behaviours correspond to the untyped test T, but do not correspond to any well-typed test, which only has access to the abstract type X and not to the concrete type *Bool*. As a result, no well-typed test can cause the action $\xrightarrow{t(a,b)}$ to be performed. We will come back to this point in Section 3.2.

2.3 Static Semantics

The static semantics for the asynchronous polymorphic π-calculus is given in Figure 3 where the domain of a typing context $\mathsf{dom}(\Gamma)$ is $\mathsf{dom}(\vec{X};\vec{n}:\vec{T}) = \{\vec{X},\vec{n}\}$, the free names of a typing context $\mathsf{fn}(\Gamma)$ are $\mathsf{fn}(\vec{X};\vec{n}:\vec{T}) = \mathsf{fn}(\vec{n})$, the free variables of a typing context $\mathsf{fv}(\Gamma)$ are $\mathsf{fv}(\vec{X};\vec{n}:\vec{T}) = \mathsf{fv}(\vec{n})$, and the free type variables of a typing context

$\mathsf{ftv}(\Gamma)$ are $\mathsf{ftv}(\vec{X};\vec{n}:\vec{T}) = \{\vec{X}\} \cup \mathsf{ftv}(\vec{T})$. We say that a typing context Δ is closed if $\mathsf{fv}(\Delta) = \mathsf{ftv}(\Delta) = \emptyset$ and moreover for any $a : T \in \Delta$ and $a : U \in \Delta$ then $T = U$. We write $\Gamma[\sigma]$ as the typing context given by $(\vec{X};\vec{n}:\vec{T})[\vec{W}/\vec{Y};\vec{m}/\vec{y}] = (\vec{X} \setminus \vec{Y};\vec{n}[\vec{m}/\vec{y}] : \vec{T}[\vec{W}/\vec{Y}])$.

This is quite a simple type system, as it does not include subtyping, bounded polymorphism, or recursive types, although we expect that such features could be added with little extra complexity.

In Section 4, we will discuss the relationship between this type system and that of Pierce and Sangiorgi. For the moment, we will just highlight one crucial non-standard point about our typing judgement: we are allowing identifiers to have more than one type in a typing context. For example:

$$X,Y;a : \updownarrow[\updownarrow[X],\updownarrow[Y]],b : \updownarrow[X],b : \updownarrow[Y] \vdash \overline{a}\langle b,b\rangle$$

To motivate the use of these mulitcontexts consider the processes

$$P \stackrel{\mathsf{def}}{=} c(X,Y;x : \updownarrow[\updownarrow[X],\updownarrow[Y]]) \,.\, x(y : \updownarrow[X],z : \updownarrow[Y]) \,.\, \overline{x}\langle y,z\rangle$$
$$Q \stackrel{\mathsf{def}}{=} \nu(a : \updownarrow[\updownarrow[\mathsf{int}],\updownarrow[\mathsf{int}]])\nu(b : \updownarrow[\mathsf{int}])\overline{c}\langle \mathsf{int},\mathsf{int};a\rangle \mid \overline{a}\langle b,b\rangle$$

which can interact as follows:

$$\begin{aligned} P \mid Q &\xrightarrow{\tau} \nu(a : \updownarrow[\updownarrow[\mathsf{int}],\updownarrow[\mathsf{int}]])(a(y : \updownarrow[\mathsf{int}],z : \updownarrow[\mathsf{int}]) \,.\, \overline{a}\langle y,z\rangle \mid \nu(b : \updownarrow[\mathsf{int}])(\overline{a}\langle b,b\rangle)) \\ &\xrightarrow{\tau} \nu(a : \updownarrow[\updownarrow[\mathsf{int}],\updownarrow[\mathsf{int}]])\nu(b : \updownarrow[\mathsf{int}])\overline{a}\langle b,b\rangle \end{aligned}$$

This interaction comes about due to the following labelled transitions from P (with appropriate matching transitions from Q):

$$\begin{aligned} P &\xrightarrow{c(\mathsf{int},\mathsf{int};a)} a(y : \updownarrow[\mathsf{int}],z : \updownarrow[\mathsf{int}]) \,.\, \overline{a}\langle y,z\rangle \\ &\xrightarrow{a(b,b)} \overline{a}\langle b,b\rangle \end{aligned}$$

Now, P typechecks as:

$$c : \updownarrow[X,Y;\updownarrow[\updownarrow[X],\updownarrow[Y]]] \vdash P$$

and we would like to find an appropriate typing for $\overline{a}\langle b,b\rangle$. The obvious typing would be to use Q's choice of concrete implementation of X and Y as int however in order to reason about P independently of Q we must choose a typing which preserves type abstraction and is independent of any choice provided by Q. To do this we use a typing which more closely resembles P's view of the interaction:

$$X,Y;c : \updownarrow[X,Y;\updownarrow[\updownarrow[X],\updownarrow[Y]]],a : \updownarrow[\updownarrow[X],\updownarrow[Y]],b : \updownarrow[X],b : \updownarrow[Y] \vdash \overline{a}\langle b,b\rangle$$

which makes a use of two different types for b in the type environment.

Pierce and Sangiorgi do not allow multiple typings for the same identifier: instead, they use *type unification* for the same purpose. In their model, the types X and Y above would be unified, and so b would just have one type $b : \updownarrow[X]$. This produces a model which is sound, but not complete, as we discuss in Section 4.

An alternative strategy to either multiple typings for variables or type unification would be subtyping with intersection types [6, 28], which ensure that meets exist in the subtype relation. Subtyping with meets are used, for example, by Hennessy and Riely [12] to ensure subject reduction. Intersection types would provide this language with pleasant properties such as principal typing, which it currently lacks, but at the cost of complexity.

3 Equivalences for Asynchronous Polymorphic Pi-Calculus

Process equivalence has a long history, including Milner's [19] bisimulation, Brookes, Hoare and Roscoe's [4] failures-divergences equivalence, and Hennessy's [11] testing equivalence. In this paper, we will follow Pierce and Sangiorgi [23] and investigate *contextual equivalence* on processes [13, 22].

Contextual equivalence has a very natural definition: it is the most generous equivalence satisfying three natural properties: *reduction closure* (that is, respecting the operational semantics), *contextuality* (that is, respecting the syntax of the language), and *barb preservation* (that is, respecting output on visible channels).

Unfortunately, although contextual equivalence has a very natural definition, it is difficult to reason about directly, due to the requirement of contextuality. Since contextuality requires processes to be equivalent in all contexts, to show contextual equivalence of P and Q, we have to show contextual equivalence of $\mathcal{C}[P]$ and $\mathcal{C}[Q]$ for any appropriately typed context $\mathcal{C}$: moreover, attempts to show this by induction on $\mathcal{C}$ break down due to reduction closure.

The problem of showing processes to be contextually equivalent is not restricted to polymorphic π-calculi, for example this problem comes up in treatments of the λ-calculus [2], monomorphic π-calculus [20] and object languages [1]. The standard solution is to ask for a *fully abstract* model, which coincides with contextual equivalence, but is hopefully more tractable.

The problem of finding fully abstract models of programming languages originates with Milner [18], and was investigated in depth by Plotkin [25] for the functional language PCF. For polymorphic languages, logical relations [27] allow for the construction of fully abstract models [24] but require an induction on type, and so break down in the presence of recursive types. Sumii and Pierce have recently shown that a hybrid of context bisimulation and logical relations [30] yields a fully abstract model in the presence of recursive types.

The monomorphic first order [20] and higher-order [29] π-calculus have quite simple fully abstract models, but to date the only known models for polymorphic π-calculus have been sound but not complete [23, 3]. We will now show that a very direct treatment of type-respecting labelled transitions generates a fully abstract bisimulation equivalence which makes no use of logical relations or type unification.

3.1 Contextual Equivalence

Process contexts are typed as follows: $\Delta \vdash \mathcal{C}[\Gamma]$ whenever $\forall(\Gamma \vdash P)\,.\,(\Delta \vdash \mathcal{C}[P])$. A typed relation on closed processes $\mathcal{R}$ is a set of triples (Γ, P, Q) such that $\Gamma \vdash P$ and $\Gamma \vdash Q$ such that Γ is closed. We will typically write $\Gamma \vDash P\ \mathcal{R}\ Q$ whenever $(\Gamma, P, Q) \in \mathcal{R}$. Given any typed relation on closed processes $\mathcal{R}$, we can define its open extension $\mathcal{R}^\circ$ to be the typed relation on processes given by $\Gamma \vDash P\ \mathcal{R}^\circ\ Q$ whenever $\Gamma[\sigma], \Delta \vDash P[\sigma]\ \mathcal{R}\ Q[\sigma]$ for any closed typing environment of the form $(\Gamma[\sigma], \Delta)$.

Definition 4 (Reduction closure). *A typed relation $\mathcal{R}$ on closed processes is reduction-closed whenever $\Delta \vDash P\ \mathcal{R}\ Q$ and $P \xrightarrow{\tau} P'$ implies there exists some Q' such that $Q \Longrightarrow Q'$ and $\Delta \vDash P'\ \mathcal{R}\ Q'$.*

$$\alpha ::= \tau \mid \nu(\vec{a} : \vec{T})c[\vec{U};\vec{b}] \mid \nu(\vec{a})\overline{c}\langle \vec{X};\vec{b} : \vec{V}\rangle \quad \text{(Typed Labels)}$$
$$C ::= (\Gamma \vdash [\sigma]P) \quad \text{(Configurations)}$$

$$\frac{P \xrightarrow{\tau} P'}{(\Gamma \vdash [\sigma]P) \xrightarrow{\tau} (\Gamma \vdash [\sigma]P')} \text{(TR-SILENT)}$$

$$\frac{\Gamma, \vec{a} : \vec{T} \vdash \overline{c}\langle \vec{U};\vec{b}\rangle \quad \{\vec{a}\} \cap \mathsf{dom}(\Gamma) = \emptyset \quad \vec{T} \text{ are generative}}{(\Gamma \vdash [\sigma]P) \xrightarrow{\nu(\vec{a}:\vec{T})c[\vec{U};\vec{b}]} (\Gamma, \vec{a} : \vec{T} \vdash [\sigma]P \mid (\overline{c}\langle \vec{U};\vec{b}\rangle[\sigma]))} \text{(TR-RECEP)}$$

$$\frac{P \xrightarrow{\nu(\vec{a}:\vec{T})\overline{c}\langle \vec{U};\vec{b}\rangle} P' \quad \Gamma \vdash c(\vec{X};\vec{x} : \vec{V}) . \mathbf{0} \quad \{\vec{a}, \vec{X}\} \cap \mathsf{dom}(\Gamma) = \emptyset}{(\Gamma \vdash [\sigma]P) \xrightarrow{\nu(\vec{a})\overline{c}\langle \vec{X};\vec{b}:\vec{V}\rangle} (\vec{X}, \Gamma, \vec{b} : \vec{V} \vdash [\vec{U}/\vec{X}, \sigma]P')} \text{(TR-OUT-W)}$$

Fig. 4. Typed Labelled Transitions $C \xrightarrow{\alpha} C'$

Definition 5 (Contextuality). *A typed relation $\mathcal{R}$ on closed processes is contextual whenever $\Gamma \vDash P \; \mathcal{R}^\circ \; Q$ and $\Delta \vdash \mathcal{C}[\Gamma]$ implies $\Delta \vDash \mathcal{C}[P] \; \mathcal{R}^\circ \; \mathcal{C}[Q]$.*

Definition 6 (Barb preservation). *A typed relation $\mathcal{R}$ on closed processes is barb-preserving whenever $\Delta \vDash P \; \mathcal{R} \; Q$ and $P \xrightarrow{\overline{a}\langle\rangle}$ implies $Q \xRightarrow{\overline{a}\langle\rangle}$.*

We can now define contextual equivalence $\cong$ as the open extension of the largest symmetric typed relation on closed processes which is reduction-closed, contextual and barb-preserving. The requirement of contextuality makes it very difficult to prove properties about contextual equivalence, and so we investigate bisimulation as a more tractable proof technique for establishing contextual equivalence.

3.2 Bisimulation

As a first attempt to find a more tractable presentation of contextual equivalence, we could use *bisimulation*. Unfortunately, as we discussed in Section 2.2, our untyped labelled transition system does not respect the type system, and so gives rise to too fine an equivalence. We therefore investigate a restricted labelled transition system which respects types: this is defined in Figure 4. The transition system is given by a relation:

$$(\Gamma \vdash [\sigma]P) \xrightarrow{\alpha} (\Gamma' \vdash [\sigma']P')$$

between configurations of the form $(\Gamma \vdash [\sigma]P)$. These comprise three constituent parts:

- P is the process being observed: after the transition, it becomes process P'.
- Γ is the *external* view of the typing context P operates in. This external view may not have complete information about the types, for example P may have exported the concrete type int as an abstract type X. Only X will be recorded in the typing context. As P exports more type information, Γ may grow to become Γ'. It is here that we make use of the multiple entries in type environments.

– σ is a type substitution, mapping the external view to the internal view. This mapping provides complete information about the types exported by P, for example int/X records that external type X is internal type int. Note that this substitution is **not** applied to P, we represent that with the alternative notation $P[\sigma]$.

There are three kinds of transitions:

– *Silent transitions* $(\Gamma \vdash [\sigma]P) \xrightarrow{\tau} (\Gamma \vdash [\sigma]P')$ which are inherited from the untyped transition system.
– *Receptivity transitions* $(\Gamma \vdash [\sigma]P) \xrightarrow{\nu(\vec{a}:\vec{T})c[\vec{U};\vec{b}]} (\Gamma, \vec{a}:\vec{T} \vdash [\sigma]P \mid (\overline{c}\langle \vec{U};\vec{b}\rangle[\sigma]))$ which allow the environment to send data to the process. We require the message to type-check, and we allow the environment to generate new names, which are recorded in the type environment. We are modelling an asynchronous language, and so processes are always input-enabled. Note that the process is sending no information to the environment, so the type substitution σ does not grow. Note also that the message is typed using the external view Γ but must have the type mapping σ applied to it for it to be mapped to the internal type consistent with P.
– *Output transitions* $(\Gamma \vdash [\sigma]P) \xrightarrow{\nu(\vec{a})\overline{c}\langle \vec{X};\vec{b}:\vec{V}\rangle} (\vec{X}, \Gamma, \vec{b}:\vec{V} \vdash [\vec{U}/\vec{X}, \sigma]P')$ which allow the process to send data to the environment. The channel being used to communicate with the environment must be typed $\updownarrow[\vec{X};\vec{V}]$, so the typing context is extended with abstract types $\vec{X}$ and the new type information $\vec{b}:\vec{V}$. This may result in more than one type being given to the same name, which is why we allow duplicate entries in typing contexts. The process P must have provided concrete implementations $\vec{U}$ of the abstract types $\vec{X}$: these are recorded in the type substitution.

To demonstrate how our typed labelled transitions can be used we return to the example above of processes L and L' and type environment Γ. We show a sequence of typed transitions from $(\Gamma \vdash []L)$ which cannot be matched by $(\Gamma \vdash []L')$:

$$(\Gamma \vdash []L) \xrightarrow{\nu(b,c,d)\overline{a}\langle X,Y;b:\updownarrow[X],b:\updownarrow[Y],c:\updownarrow[Y],d:Y\rangle} (\Gamma' \vdash [\sigma]c(y:\updownarrow[T]).\overline{\mathsf{fail}}\langle\rangle)$$

where σ is $[T,T/X,Y]$ and Γ' is $X,Y,\Gamma,b:\updownarrow[X],b:\updownarrow[Y],c:\updownarrow[Y],d:X$. At this point we would like to use Rule TR-RECEP to provide a message on channel c to facilitate a communication, however, there is no name of the appropriate type listed in Γ' and the restriction to generative types for the fresh names means that this cannot yet be done. However, note the following transitions:

$$\begin{array}{rl}(\Gamma' \vdash [\sigma]c(y:\updownarrow[T]).\overline{\mathsf{fail}}\langle\rangle) & \xrightarrow{b[d]} \quad (\Gamma' \vdash [\sigma]c(y:\updownarrow[T]).\overline{\mathsf{fail}}\langle\rangle \mid \overline{b}\langle d\rangle) \\ & \xrightarrow{\overline{b}\langle d\rangle} \quad (\Gamma', d:Y \vdash [\sigma]c(y:\updownarrow[T]).\overline{\mathsf{fail}}\langle\rangle) \\ & \xrightarrow{c[d]} \quad (\Gamma', d:Y \vdash [\sigma]c(y:\updownarrow[T]).\overline{\mathsf{fail}}\langle\rangle \mid \overline{c}\langle d\rangle) \\ & \xRightarrow{\overline{\mathsf{fail}}\langle\rangle}\end{array}$$

in which the second type listed for b in Γ' is used to justify the $\overline{b}\langle d\rangle$ transition. These transitions serve to mimic the typecasting and subsequent use of the extruded name d by a testing context which are crucial to distinguishing L and L'.

We now formalise our notion of bisimulation equivalence. A typed relation on closed configurations $\mathcal{R}$ is a set of 5-tuples $(\Gamma, \sigma, P, \rho, Q)$ such that $\Gamma[\sigma] \vdash P$ and $\Gamma[\rho] \vdash Q$ and both $\Gamma[\sigma]$ and $\Gamma[\rho]$ are closed. For convenience we will write $\Gamma \vDash [\sigma]P \; \mathcal{R} \; [\rho]Q$ whenever $(\Gamma, \sigma, P, \rho, Q) \in \mathcal{R}$.

Definition 7 (Bisimulation). *A simulation $\mathcal{R}$ is a typed relation on closed configurations such that if $\Gamma \vDash [\sigma]P \; \mathcal{R} \; [\rho]Q$ and $(\Gamma \vdash [\sigma]P) \xrightarrow{\alpha} (\Gamma' \vdash [\sigma']P')$ then we can show $(\Gamma \vdash [\rho]Q) \stackrel{\widehat{\alpha}}{\Longrightarrow} (\Gamma' \vdash [\rho']Q')$ for some $\Gamma' \vDash [\sigma']P' \; \mathcal{R} \; [\rho']Q'$. A bisimulation is a simulation whose inverse is also a simulation. Let $\approx$ be the largest bisimulation.*

We are now in position to show full abstraction of bisimulation for contextual equivalence, and so provide a tractable model of polymorphic π-calculus.

3.3 Soundness of Bisimulation for Contextual Equivalence

The difficult property to show is that bisimulation is a congruence: from this it is routine to establish that bisimulation implies contextual equivalence. Showing congruence for bisimulation is a well-established problem for process languages, going back to Milner [19]. In the case of polymorphic π, the problem is in showing that bisimulation is preserved by parallel composition. We do this by constructing a candidate bisimulation:

$$\begin{array}{rl} \Gamma \vDash [\sigma]P \mid R[\sigma] \; \mathcal{R} \; [\rho]Q \mid R[\rho] & \text{whenever } \Gamma \vDash [\sigma]P \approx [\rho]Q \\ & \text{and } \Gamma \vdash R \\ & \text{and } \sigma \text{ and } \rho \text{ are type substitutions} \end{array}$$

and then showing that this is a bisimulation (up to some technicalities which we shall elide for the moment). This has a routine proof, except for one case, which is when $R[\sigma] \longrightarrow R'[\sigma]$. It is straightforward to establish that type substitutions do not influence reduction, and so we have $R[\rho] \longrightarrow R'[\rho]$, and all that remains is to show that $\Gamma \vDash [\sigma]P \mid R'[\sigma] \; \mathcal{R} \; [\rho]Q \mid R'[\rho]$. Unfortunately, this is not directly possible, due to the requirement that $\Gamma \vdash R'$. If we had a subject reduction result for open processes, then this would be routine, but this result is not true due to channels with multiple types:

$$\begin{array}{rcl} \overline{a}\langle c\rangle \mid a(x:Y) \,.\, \overline{b}\langle x\rangle & \longrightarrow & \mathbf{0} \mid \overline{b}\langle c\rangle \\ X, Y; a : \updownarrow[X], a : \updownarrow[Y], b : \updownarrow[Y], c : X & \vdash & \overline{a}\langle c\rangle \mid a(x:Y) \,.\, \overline{b}\langle x\rangle \\ X, Y; a : \updownarrow[X], a : \updownarrow[Y], b : \updownarrow[Y], c : X & \not\vdash & \mathbf{0} \mid \overline{b}\langle c\rangle \end{array}$$

Pierce and Sangiorgi's technique for dealing with this problem is to introduce type unification to ensure that every channel has a unique type. Unfortunately, as we will discuss in Section 4, the resulting semantics is incomplete. Instead of using such unifications, we observe that in any case where subject reduction fails, it does so because of communication on a visible channel: if the channel was hidden by a ν-binder, then it would have only one type, and so subject reduction holds. We therefore observe that in the cases where subject reduction fails to hold, there must be a pair of matching visible reductions which caused the communication.

Proposition 1 (Open subject reduction). *If* $\Gamma \vdash P$ *and* $P \xrightarrow{\tau} P''$ *then either:*

1. $\Gamma \vdash P''$, *or*
2. $P \xrightarrow{\nu(\vec{a}:\vec{T})\overline{c}\langle\vec{U};\vec{b}\rangle} \xrightarrow{c(\vec{X};\vec{b})} P'$ *where* $P'' \equiv (\nu(\vec{a}:\vec{T})P')[\vec{U}/\vec{X}]$.

In the example (up to structural equivalence):

$$\begin{array}{rcl}
\overline{a}\langle c\rangle \mid a(x:Y).\,\overline{b}\langle x\rangle & \xrightarrow{\overline{a}\langle c\rangle} & \mathbf{0} \mid a(x:Y).\,\overline{b}\langle x\rangle \\
 & \xrightarrow{a(c)} & \mathbf{0} \mid \overline{b}\langle c\rangle \\
X,Y;a:\updownarrow[X],a:\updownarrow[Y],b:\updownarrow[Y],c:X & \vdash & \overline{a}\langle c\rangle \mid a(x:Y).\,\overline{b}\langle x\rangle \\
X,Y;a:\updownarrow[X],a:\updownarrow[Y],b:\updownarrow[Y],c:X,c:Y & \vdash & \mathbf{0} \mid a(x:Y).\,\overline{b}\langle x\rangle \\
X,Y;a:\updownarrow[X],a:\updownarrow[Y],b:\updownarrow[Y],c:X,c:Y & \vdash & \mathbf{0} \mid \overline{b}\langle c\rangle
\end{array}$$

The crucial point is that these extra transitions by the testing context correspond to complementary typed transitions by the process such that, after the visible $\overline{a}\langle c\rangle$ output action, the typing context Γ is extended with $c:Y$. The problematic residual of the test term R' ($\mathbf{0} \mid \overline{b}\langle c\rangle$ in the example) can now be typed in this extended Γ and the bisimulation argument can be completed.

Theorem 1 (Bisimulation is a congruence). *If* $\Gamma \vDash P \approx^{\circ} Q$ *then* $\Delta \vDash \mathcal{C}[P] \approx^{\circ} \mathcal{C}[Q]$ *for any* $\Delta \vdash \mathcal{C}[\Gamma]$.

Theorem 2 (Soundness of bisimulation for contextual equivalence). *If* $\Gamma \vDash P \approx^{\circ} Q$ *then* $\Gamma \vDash P \cong Q$.

3.4 Completeness of Bisimulation for Contextual Equivalence

The proof of soundness for bisimulation required some non-standard techniques. In comparison, the proof of completeness is quite straightforward, and follows the usual *definability* argument [11, 9, 15] of showing that for every visible action α, we can find a process R which exactly tests for the ability to perform α. Once we have established definability, completeness follows in a straightforward fashion.

Theorem 3 (Completeness of bisimulation for contextual equivalence). *If* $\Gamma \vDash P \cong Q$ *then* $\Gamma \vDash P \approx^{\circ} Q$.

4 Comparison with Pierce and Sangiorgi

In this paper, we have shown that weak bisimulation is fully abstract for observational equivalence for an asynchronous polymorphic π-calculus. This is almost enough to settle the open problem set by Pierce and Sangiorgi [23] of finding a fully abstract semantics for their polymorphic π-calculus. There are, however, some differences between their setting and ours, most of which we believe to be routine, with one important exception: the type rule for if-then-else.

4.1 Minor Differences

The minor differences between our polymorphic π-calculus and theirs are:

1. We are considering weak bisimulation rather than strong bisimulation.
2. Since we are considering weak bisimulation, we have not included $P+Q$ in our language of processes. We expect that this could be handled in the usual fashion, by defining observational equivalence on processes in the style of Milner [19].
3. We have treated an asynchronous rather than a synchronous language, since the soundness result follows more naturally for the resulting asynchronous transition system. We expect that a fully abstract bisimulation for a synchronous language can be given by adding transitions for synchronous input as well as receptivity:

$$\frac{P \xrightarrow{c(\vec{U};\vec{b})} P' \quad \Gamma, \vec{a} : \vec{T} \vdash \overline{c}\langle \vec{U};\vec{b}\rangle \quad \{\vec{a}\} \cap \mathsf{dom}(\Gamma) = \emptyset \quad \vec{T} \text{ are generative}}{(\Gamma \vdash [\sigma]P) \xrightarrow{\nu(\vec{a}:\vec{T})c(\vec{U};\vec{b})} (\Gamma, \vec{a} : \vec{T} \vdash [\sigma]P')} \text{(TR-IN)}$$

   Note that the label used here for synchronous input is distinct from the label used for receptivity.
4. We have used a variable-name distinction, and so have used Honda and Yoshida's definition of observational equivalence [13]. See [8] for a discussion of this issue.
5. Our type system keeps track explicitly of free type variables, rather than treating them implicitly: this makes some of the book-keeping easier, at the cost of some additional syntactic overhead.

We do not believe that these differences are substantial.

4.2 Major Difference: Typing If-Then-Else

However, there is one important difference between our language and Pierce and Sangiorgi's, even though it may appear at first sight to be a minor point: the type rule for if-then-else. In their paper, a strong type rule is given:

$$\frac{\Gamma \vdash n : T \quad \Gamma \vdash m : T \quad \Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash \mathsf{if}\, n = m \,\mathsf{then}\, P \,\mathsf{else}\, Q} \text{(T-TEST-S)}$$

In our work, the weaker type rule T-TEST-W is used, which allows n and m to have different types. Note that in a language with subtyping and a top type, these rules are equivalent, since we can always choose T to be the top type, and use subsumption to derive T-TEST-W from T-TEST-S. In the absence of subtyping, however, the rule T-TEST-W allows more processes to typecheck, so raises the expressive power of tests, and hence makes observational equivalence finer. For example:

$$P \stackrel{\mathsf{def}}{=} \nu(b : \updownarrow[\mathsf{int}])\nu(c : \updownarrow[\mathsf{string}])\overline{a}\langle \mathsf{int}, \mathsf{string}; b, c\rangle$$
$$Q \stackrel{\mathsf{def}}{=} \nu(b : \updownarrow[\mathsf{int}])\overline{a}\langle \mathsf{int}, \mathsf{int}; b, b\rangle$$

As long as $a : \updownarrow[X,Y;\updownarrow[X],\updownarrow[Y]]$ these processes cannot be distinguished by any test which uses the type rule T-TEST-S, but they can be distinguished by:

$$R \stackrel{\text{def}}{=} a(X,Y;x:\updownarrow[X],y:\updownarrow[Y]).\,\text{if } x = y \text{ then } \overline{d}\langle\rangle$$

which typechecks using type rule T-TEST-W. In fact, there is a third possible type rule for if-then-else, which makes use of type unification:

$$\frac{\Gamma \vdash n : T \quad \Gamma \vdash m : U \qquad \mathsf{mgu}(T,U) = \sigma \Rightarrow \Gamma[\sigma] \vdash P[\sigma] \quad \Gamma \vdash Q}{\Gamma \vdash \text{if } n = m \text{ then } P \text{ else } Q} \text{ (T-TEST-U)}$$

where $\mathsf{mgu}(T,U)$ builds the most general type substitution σ such that $T[\sigma] = U[\sigma]$. This type rule is strictly weaker than T-TEST-W, and raises the expressive power of tests even further, and hence makes observational equivalence even finer. For example:

$$P \stackrel{\text{def}}{=} \nu(c:\updownarrow[\mathsf{int},\mathsf{string}])\nu(d:\updownarrow[\mathsf{int}])\overline{a}\langle \mathsf{int},\mathsf{string};c,d\rangle.\,\overline{b}\langle \mathsf{string};c\rangle.\,d(x:\mathsf{int}).\,\overline{e}\langle x\rangle$$
$$Q \stackrel{\text{def}}{=} \nu(c:\updownarrow[\mathsf{int},\mathsf{string}])\nu(d:\updownarrow[\mathsf{int}])\overline{a}\langle \mathsf{int},\mathsf{string};c,d\rangle.\,\overline{b}\langle \mathsf{string};c\rangle$$

As long as $a : \updownarrow[X,Y;\updownarrow[X,Y],\updownarrow[X]], b : \updownarrow[Z;\updownarrow[\mathsf{int},Z]]$ and $e : \updownarrow[\mathsf{int}]$, these processes cannot be distinguished by any test which uses T-TEST-W, but they can be distinguished by:

$$R \stackrel{\text{def}}{=} a(X,Y;x:\updownarrow[X,Y],y:\updownarrow[X]).\,b(Z;z:\updownarrow[\mathsf{int},Z]).\,\text{if } x = z \text{ then } \overline{y}\langle 5\rangle$$

which typechecks using type rule T-TEST-U. We have that:

- The type rule T-TEST-W has a matching fully abstract bisimulation equivalence $\approx$, which for purpose of this discussion we shall refer to as $\approx_{\mathsf{w}}$.
- The type rule T-TEST-S has a matching fully abstract bisimulation equivalence $\approx_{\mathsf{s}}$.
- The type rule T-TEST-U has a matching fully abstract bisimulation equivalence $\approx_{\mathsf{u}}$.

Moreover:

- We have inclusions on these equivalences: if $\Gamma \vDash P \approx_{\mathsf{w}} Q$ then $\Gamma \vDash P \approx_{\mathsf{s}} Q$ for any $\Gamma \vdash_{\mathsf{s}} P$ and $\Gamma \vdash_{\mathsf{s}} Q$ (and similarly for $\approx_{\mathsf{u}}$ and $\approx_{\mathsf{w}}$).
- The above examples show that the inclusions are strict: we have $\Gamma \vDash P \not\approx_{\mathsf{w}} Q$ and $\Gamma \vDash P \approx_{\mathsf{s}} Q$ for some $\Gamma \vdash_{\mathsf{s}} P$ and $\Gamma \vdash_{\mathsf{s}} Q$ (and similarly for $\approx_{\mathsf{u}}$ and $\approx_{\mathsf{w}}$).
- The type rule for if-then-else used by Pierce and Sangiorgi is T-TEST-S.
- Pierce and Sangiorgi's bisimulation is the strong, synchronous version of $\approx_{\mathsf{u}}$.

Hence, since synchrony and weak bisimulation play no role in the above examples, we have a resolution of Pierce and Sangiorgi's conjecture:

- Pierce and Sangiorgi's polymorphic bisimulation is sound, but not complete, for their polymorphic π-calculus.

These arguments are formalised in [16].

5 Conclusions

This paper gives the first fully abstract semantics for a polymorphic process language. Moreover the semantics is extremely straightforward: the only nonstandard part of the presentation is that names are given more than one type in a type environment. This corresponds to the ability for a polymorphic program to be sent the same channel at multiple different types. In contrast to polymorphic λ-calculi, polymorphic π-calculi have the ability to compare names for syntactic equality, and so there is an internal test which can detect when the same name has been given multiple different types.

We believe that the techniques given in this paper are quite robust (for example there are no uses of type induction) and could be scaled with little difficulty to larger type systems with features such as subtyping, F-bounded polymorphism, and recursive types. Moreover, object languages such as the ς-calculus support object equality, and so we believe that adapting our previous fully abstract semantics [14] for objects [1] to deal with generic objects would also be possible.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North Holland, 1984.
3. M. Berger, K. Honda, and N. Yoshida. Genericity and the pi-calculus. In *Proc. Int. Conf. Foundations of Software Science and Computer Structures (FoSSaCs)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
4. S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.
5. P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proc. Int. Conf. Functional Programming Languages and Computer Architecture (FPCA)*, pages 273–280. ACM Press, 1989.
6. M. Coppo and M. Dezani-Ciancaglini. A new type-assignment for λ-terms. *Archiv Math. Logik*, 19:139–156, 1978.
7. Microsoft Corporation. ECMA and ISO/IEC c# and common language infrastructure standards, 2004. http://msdn.microsoft.com/net/ecma/.
8. C. Fournet and G. Gonthier. A hierarchy of equivalences for asynchronous calculi. In *Proc. Int. Conf. Automata, Languages and Programming (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
9. C. Fournet, G. Gonthier, J-J. Levy, L. Maranget, and D. Remy. A calculus of mobile agents. In *Proc. Int. Conf. Concurrency Theory (CONCUR)*, volume 1119 of *Lecture notes in computer science*. Springer-Verlag, 1996.
10. J-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, 1989.
11. M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.
12. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173(1):82–120, 2002.
13. K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995.
14. A. S. A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In *Proc. IEEE Logic In Computer Science*, pages 101–112. IEEE Press, 2002. Full version to appear in *Theoretical Computer Science*.

15. A. S. A. Jeffrey and J. Rathke. Contextual equivalence for higher-order pi-calculus revisited. In *Proc. Mathematical Foundations of Programming Semantics*, Electronic Notes in Computer Science. Elsevier, 2003.
16. A. S. A. Jeffrey and J. Rathke. Full abstraction for polymorphic pi-calculus. Online edition with proofs, http://www.fabfac.org/, 2005.
17. Sun Microsystems. Release notes Java 2 platform standard edition development kit 5.0, 2004. http://java.sun.com/j2se/1.5.0/relnotes.html.
18. R. Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–22, 1977.
19. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
20. R. Milner. *Communication and mobile systems: the π-calculus*. Cambridge University Press, 1999.
21. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Part I + II. *Information and Computation*, 100(1):1–77, 1992.
22. R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proc. Int. Conf. Automata, Languages and Programming (ICALP)*, volume 623 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
23. B. C. Pierce and D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *J. ACM*, 47(3):531–584, 2000.
24. A. M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.
25. G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
26. J. C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing*, 83:513–523, 1983.
27. J. C. Reynolds. An introduction to logical relations and parametric polymorphism (abstract). In *Proc. ACM Symp. Principles of Programming Languages*, pages 155–156. ACM Press, 1993.
28. J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.
29. D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, 1993.
30. E. Sumii and B. C. Pierce. A bisimulation for type abstraction and recursion. In *Proc. ACM Symp. Principles of Programming Languages*, 2005. To appear.
31. P. Wadler. Theorems for free! In *Proc. Int. Conf. Functional Programming Languages and Computer Architecture (FPCA)*, pages 347–359. ACM Press, New York, 1989.