Pict: A Programming Language Based on the Pi-Calculus

Benjamin C. Pierce **David N. Turner**

Indiana University
CSCI Technical Report #476

March 19, 1997

Abstract

The π-calculus offers an attractive basis for concurrent programming. It is small, elegant, and well studied, and supports (via simple encodings) a wide range of high-level constructs including data structures, higher-order functional programming, concurrent control structures, and objects. Moreover, familiar type systems for the λ-calculus have direct counterparts in the π-calculus, yielding strong, static typing for a high-level language using the π-calculus as its core. This paper describes Pict, a strongly-typed concurrent programming language constructed in terms of an explicitly-typed π-calculus core language.

Dedicated to Robin Milner on the occasion of his 60th birthday.

1 Introduction

Milner, Parrow, and Walker's π-calculus [MPW92, Mil91] generalizes the channel-based communication of CCS and its relatives by allowing channels to be passed as data along other channels. This extension introduces an element of *mobility*, enabling the specification and verification of concurrent systems with dynamically evolving communication topologies. Channel mobility leads to a surprising increase in expressive power, yielding a calculus capable of describing a wide variety of high-level concurrent features while retaining a simple semantics and tractable algebraic theory.

A similar combination of simplicity and expressiveness has made the λ-calculus both a popular object of theoretical investigation and an attractive basis for sequential programming language design. By analogy, then, one may wonder what kind of high-level programming language can be constructed from the π-calculus.

$$\frac{\text{ML, Haskell, \ldots}}{\lambda\text{-calculus}} = \frac{?}{\pi\text{-calculus}}$$

A number of programming language designs have combined π-calculus-like communication with a functional core language, but none have gone so far as to take communication as the sole mechanism of computation. The primary motivation of the Pict project, begun at the University of Edinburgh in 1992, was to design and implement a high-level concurrent language purely in terms of the π-calculus primitives [PT97, Pie97].

Compiling a language based solely on communicating processes raises challenging code generation problems. To achieve acceptable performance for realistic applications, a π-calculus compiler must implement process creation, context switching, and communication on channels extremely efficiently, since these operations are the fundamental computational mechanism used in the π-calculus and, for example, are at least as pervasive as function calls in a functional language.

Another goal of the Pict project was to explore the practical applicability of our earlier theoretical work on type systems for the π-calculus [PS93, Tur96] and on λ-calculus type systems with subtyping [PT94, HP95, PS97]. In particular, in [PT94] we proposed a powerful combination of subtyping and polymorphism as a basis for statically typed object-oriented programming in functional languages; equipping Pict with a similar type system provides a testbed for experiments with statically typed concurrent objects. Using such a powerful type system raises other important issues such as typechecking algorithms, efficiency of typechecking, and type inference.

The questions motivating the Pict project, then, can be summarized as follows: (1) What is it like to program in the π-calculus? What kind of high-level language can be built on it? (2) What kinds of concurrent objects arise in this setting? (3) Can the π-calculus be implemented efficiently? (4) Can we design a practical type system for the π-calculus combining subtyping and higher-order polymorphism?

In this paper, we offer our responses to these questions (concentrating on (1) and (4), since (2) has been addressed in detail in [PT95] and (3) in [Tur96]), and survey the current state of the Pict language. Section 2 defines the syntax and operational semantics of the core language and discusses some points where it differs from the theoretical π-calculus of Milner, Parrow, and Walker. Section 3 presents a type system for the core language, incorporating channel types, subtyping, record types, higher-order polymorphism, and simple recursive types. Section 4 constructs the full Pict language by means of translations into the core. Section 5 offers concluding remarks and directions for further research.

Please note that we do not attempt to give a definitive description of Pict here, since Pict is an experimental language and is therefore subject to relatively frequent changes in design. Instead, we give an overview of the main decisions we have made in the design of Pict. For a complete description of the current version of Pict, please consult the formal definition [PT97], which is kept up to date with the current version of the compiler.

2 The Core Language

We now proceed to a rigorous definition of the syntax and semantics of the core language: an asynchronous, choice-free fragment of the π-calculus enriched with records and pattern matching.

2.1 The Pi-Calculus

To aid comparison, we begin with a brief review of the pure π-calculus of Milner, Parrow, and Walker. More details can be found in the original π-calculus papers [MPW92] and in Milner's tutorial [Mil91].

The computational world modeled by the π-calculus contains just two kinds of entities: processes and channels. Processes, sometimes called agents, are the active components of a system; they interact by synchronous rendezvous on channels, also called names or ports. When two processes synchronize, they exchange a single data value, which is itself a channel. The output process $\bar{x}y.e_1$ sends y along x and then, after the output has completed, continues as e_1. Conversely, the input process $x(z).e_2$ waits until a value is received along x, substitutes it for the bound variable z, and continues as e_2. The parallel composition of these processes, written $\bar{x}y.e_1 \mid x(z).e_2$, may thus synchronize on x, yielding the derivative $e_1 \mid \{z \mapsto y\}e_2$.

Fresh channels are introduced by the restriction operator ν. The expression $(\nu x)e$ creates a fresh channel x with lexical scope e. For example, writing $(\nu x)\,(\bar{x}y.e_1 \mid x(z).e_2)$ localizes the channel x, ensuring that no other process can interfere with the communication on x.

The expression $e_1 + e_2$ denotes an external choice between e_1 and e_2: either e_1 is allowed to proceed and e_2 is discarded, or vice versa. For example, the process $\bar{x}y.e_1 \mid (x(z).e_2 + x(w).e_3)$ can reduce either to $e_1 \mid \{z \mapsto y\}e_2$ or to $e_1 \mid \{w \mapsto y\}e_3$. The nullary choice, written $\mathbf{0}$, is inert.

Infinite behavior in π-calculus is introduced by the replication operator $!e$, which informally denotes an arbitrary number of copies of e running in parallel. This operator replaces the equivalent, but more complex, mechanism of mutually-recursive process definitions.

Some variants of the π-calculus include a matching operator $[x = y]e$, which allows e to proceed if x and y are the same channel.

2.2 Core Language Design Issues

The core language of Pict differs from the π-calculus by some trivial extensions and some more important restrictions.

2.3 Primitive Values

Like most high-level programming languages, Pict provides special syntax for a few built-in types: booleans, characters, strings, and integers. Adding such syntax does not change the fundamental character of the language at all, since all these types of data can easily be encoded as processes [Mil91]. To give the compiler maximum freedom to implement primitive values efficiently, the types `Bool`, `Char`, `String`, and `Int` are *abstract*: they reveal nothing about how values of these types are represented. Instead, built-in channels are provided for performing common operations. For example, `12` is a built-in value of type `Int` and the addition operation is represented as a built-in channel `+`. To add two numbers, we send them along the channel `+` together with a result channel `r`, and then listen on `r` for the result of the calculation. Built-in channels are also provided for interacting with the environment; for example, the channel `print` is used to send strings to the standard output stream.

2.3.1 Records and Pattern Matching

A key choice in the design of Pict has been to define as much of the language as possible in terms of encodings. Section 4 describes many such derived forms, including, for example, function abstraction and application. This style of definition means that we need only give operational semantics and typing rules for the core language; the rules for functions arise from the translation.

The pure π-calculus can easily encode "polyadic" communication, in which several channels are exchanged during a single communication [Mil91]. Similar encodings can be used for data structures such as records. However, such encodings do not always give rise to useful derived typing rules. In particular, when we started the Pict design, there were no type systems for the pure, monadic π-calculus (although more recent work on linear types [KPT96] may lead to such type systems). Therefore, we begin from a slightly more structured core language, which admits a simple, structural type system — just as typed functional languages such as ML and Haskell are typically based on a λ-calculus extended with basic data constructors.

2.3.2 Asynchrony

The fundamental communication primitive in π-calculus and most of its relatives is *synchronous rendezvous*, in which both sender and receiver are blocked until the communication occurs. For example, in the π-calculus expression $\bar{x}y.e_1 \mid x(z).e_2$, the expression e_1 cannot proceed until the output on x has completed; similarly, the expression e_2 cannot proceed until a value z has been received along x.

The fact that output is synchronous enables a sending process to tell when its message has been received by another process. Unfortunately, depending on this information precludes a number of useful programming idioms involving buffering, delegation, and reordering of requests. For example, if a client and a server are sometimes run on separate machines, we may need to add a surrogate server process on the client's machine that forwards requests to the server's machine. Now a synchronization on the client's request channel indicates only that the surrogate, not the server, has received the request.

We allow only asynchronous output in Pict; this amounts to restricting the "continuation" of each output expression to the null process. The programmer must send an explicit acknowledgement or result to inform the client that its request has been processed, thereby eliminating any possible sensitivity to buffering or reordering of requests.

The investigation of asynchronous process calculi was initiated by Honda, Tokoro, and Yoshida [HT91, HY94] and Boudol [Bou92]. Amadio, Castellani and Sangiorgi [ACS96] have more recently shown how several technical aspects of observational equivalence are simplified in the asynchronous case.

2.3.3 No Choice

Early versions of the π-calculus used a completely unrestricted choice operator: in the expression $e_1 + e_2$, the branches e_1 and e_2 could be arbitrary processes. More recent presentations, for example [Mil91], use a more constrained operator called *guarded choice*, where e_1 and e_2 must be input expressions, output expressions, or choice expressions. Guarded choice is easier to formalize (especially in the context of a reduction semantics, such as the one presented in Sections 2.6 to 2.8) and appears to capture all cases of practical interest.

In an asynchronous language, guarded choice should be restricted still further, since an asynchronous output in a choice is sensitive to buffering: $(\bar{w}y + e)$ can only discard e if a process reads from w, while $(\bar{x}y + e) \mid x(z).\bar{w}z$ can reduce spontaneously to $\bar{w}y$.

In Pict we go a step further, dropping the choice operator altogether. This simplifies both formal semantics and implementation, and has little effect on the expressiveness of the language, since input-only choice is easy to implement as a library module [PT95, NP96]. (This has some cost in syntactic convenience, and some benefit in flexibility. Our library actually implements a subset of Reppy's *events* [Rep91], allowing the branches of a choice to be manipulated as data.) In fact, most Pict programs use simpler mechanisms such as locks and semaphores (cf. Section 4.6) for basic synchronisation tasks. For controlling concurrent execution of methods in objects, one can use a different library implementing a more specialized operator called *replicated choice* [PT95].

2.3.4 Replicated Input, No Matching

The Pict core language makes two further simplifications of the pure π-calculus. First, we restrict replication to replicated input expressions. This variant has the same formal power as full replication, but has a simpler semantics and is closer to a realistic implementation. Second, we omit the matching operator, since its main functions (encoding conditional expressions and tracking side-conditions in axiomatizations of behavioral equivalences) are either subsumed by other features in Pict or irrelevant in the context of programming.

2.3.5 Ascii Notation

Besides the more substantive changes discussed above, Pict substitutes a slightly heavier ascii concrete syntax for the more mathematical π-calculus notation:

π-calculus	Pict	
$\bar{x}y.\mathbf{0}$	`x!y`	asynchronous output
$x(y).e$	`x?y = e`	input prefix
$e_1 \mid e_2$	`(e`$_1$ ` | e`$_2$`)`	parallel composition
$(\nu x)e$	`(new x e)`	channel creation
$!x(y).e$	`x?*y = e`	replicated input

2.4 Core Language Syntax

We now define the Pict core language syntax. Further details, such as lexical analysis rules, can be found in the Pict language definition [PT97]. The possible forms of each production are listed on successive lines. Keywords are set in typewriter font. An expression of the form $X \ldots X$ denotes a list of zero or more occurrences of X. The expression $\langle$*empty*$\rangle$ denotes an empty production.

The entities that can be communicated on channels are called *values*. They include variables, records of values, package values, rectype values, and constants.

Val	=	*Id*	Variable
		`[` *Label Val* `...` *Label Val* `]`	Record
		`{` *Type* `}` *Val*	Polymorphic package
		`( rec : T` *Val* `)`	Rectype value
		String	String constant
		Char	Character constant
		Int	Integer constant
		Bool	Boolean constant
Label	=	$\langle$*empty*$\rangle$	Anonymous label
		Id `=`	Explicit label

There are no channel constants, only variables ranging over channels (note, however, that variables can range over any kind of value, not just channels). Record values generalise tuple values (since the labels in a record are optional).

Rectype values help the typechecker determine the types of recursive data structures; package values are part of the mechanism used to implement polymorphism in Pict. We defer the description of these to Sections 3.2 and 3.5.

Values can be decomposed by means of *patterns*. A variable pattern `x:T` binds the variable `x`. A package pattern `{X<T}p` binds the type variable `X` plus whatever variables are bound in `p`. A layered pattern `x:T@p` binds the variable `x` plus whatever variables are bound in `p`. All the variables bound by a pattern must be pairwise distinct.

Pat	=	*Id* : *Type*	Variable pattern
		_ : *Type*	Wildcard pattern
		Id : *Type* @ *Pat*	Layered pattern
		[*Label Pat* ... *Label Pat*]	Record pattern
		{ *Id* < *Type* } *Pat*	Package pattern
		(`rec` : T *Pat*)	Rectype pattern

Note that all bound variables (and wildcards) are explicitly typed. In practice, many of these type annotations can be inferred automatically by the Pict compiler. A layered pattern can be used to bind a variable to a value at the same time as decomposing the value. For example, matching the pattern `x@[y z]` against the value `[33 true]` binds `x` to `[33 true]`, `y` to `33`, and `z` to `true`. We defer the description of rectype and package patterns to Sections 3.2 and 3.5.

A process prefixed by a pattern is called an *abstraction*. Introducing a separate syntactic class of abstractions leaves room for later expansion. We make use of this in the full language to allow higher-order functions to appear wherever process abstractions are allowed (cf. Section 4.5).

Abs	=	*Pat* = *Proc*	Process abstraction

In an abstraction `p = e`, variable occurrences in `p` are binders with scope `e`.

The basic forms of processes are output atoms, input prefixes, parallel compositions, processes prefixed by declarations, and conditional processes.

Proc	=	*Val* ! *Val*	Output atom
		Val ? *Abs*	Input prefix
		Val ?* *Abs*	Replicated input prefix
		(*Proc* | *Proc*)	Parallel composition
		(*Dec Proc*)	Local declaration
		`if` *Val* `then` *Proc* `else` *Proc*	Conditional

Arbitrary values must be allowed to the left of `!` and `?` so that substitution is a total operation (cf. Section 2.7). Our type system guarantees that these values can only evaluate to channel names.

Finally, a `new` declaration introduces a new channel. Again, we make declarations a separate syntactic category to leave room for growth.

Dec	=	`new` *Id* : *Type*	Channel creation

The expression `(new x:T e)` binds `x` with scope `e`. Note that new channels are always annotated with explicit types.

2.5 Example

We now present some simple examples of Pict core language programs. In Section 3.2 we show how to annotate the examples with appropriate explicit type information. (Section 4.6 uses the additional features of the full language to express these examples much more concisely!)

The following process implements a "cons-cell server" which, when sent a triple `[hd tl r]`, constructs a process encoding a cons cell (with head `hd` and tail `tl`), and returns the address of the cons cell along the result channel `r`.

```
cons?*[hd tl r]  =  (new l (r!l | l?*[n c] = c![hd tl]))
```

Upon receiving a triple `[hd tl r]`, we first create a new channel `l` (`l` can be thought of as the location of the cons cell). Then, in parallel, we return `l` along the result channel `r` and run the process `l?*[n c] = c![hd tl]`. This process responds to messages sent along `l` by sending `hd` and `tl` along `c`.

The following process behaves similarly, except that it constructs a nil, or empty, list. Upon receiving a tuple `[r]` containing just a result channel `r`, we create a new channel `l`. Then, in parallel, we return `l` along the result channel `r` and run the process `l?*[n c] = n![]`. This process responds to messages sent along `l` by sending the trivial value `[]` back along `n`.

```
nil?*[r]  =  (new l (r!l | l?*[n c] = n![]))
```

The following program fragment illustrates how we can interact with `nil` and `cons` to build a list containing the number 33. We first create a fresh result channel `r1` and send it along the `nil` channel. In parallel, we wait for `nil`'s reply to be sent along `r1`, binding the resulting value to `e`. We then create a second result channel `r2` and send the tuple `[33 e r2]` to `cons`. This has the effect of building a cons cell whose head is 33 and tail is `e`. The location of the new cell is returned along `r2`.

```
(new r1 (nil![r1] |
         r1?e = (new r2 (cons![33 e r2] |
                          r2?l = ...))))
```

We can interrogate our list by sending a pair of channels `[n c]` along the channel `l`. By convention, an empty list will reply on `n` (by sending the trivial value `[]`), while a cons cell will reply on `c` (by sending a pair of the head and tail of the list). The following process therefore executes the expression `e` if `l` is the empty list, and `f` if `l` is a cons cell (in which case `hd` and `tl` will become bound to the head and tail of `l`):

```
(new n (new c (l![n c]  |  n?[] = e |  c?[hd tl] = f)))
```

2.6 Structural Congruence

In discussing these examples, we have appealed to an informal understanding of how Pict expressions behave. It is now time to make this understanding precise. Following [Mil91], the operational semantics of Pict programs is presented in two steps. First, we define a *structural congruence* relation $\mathtt{e}_1 \equiv \mathtt{e}_2$; this relation captures the fact that, for example, the order of the branches in a parallel composition has no effect on its behavior. Next, we define a *reduction relation* $\mathtt{e}_1 \rightarrow \mathtt{e}_2$, specifying how processes evolve by communication.

Structural congruence plays an important technical role as a device for simplifying the statement of the reduction relation. For example, we intend that the processes `(x!v | x?y = e)` and `(x?y = e | x!v)` both reduce to $\{\mathtt{y} \mapsto \mathtt{v}\}\mathtt{e}$. Since these two are structurally congruent, it suffices to write the reduction rule only for the first case and to stipulate, in general, that if `e` contains some possibility of communication then any expression structurally congruent to `e` has the same possible behavior.

The first two structural congruence rules state that parallel composition is commutative and associative.

$$(\mathtt{e}_1 \mid \mathtt{e}_2) \equiv (\mathtt{e}_2 \mid \mathtt{e}_1) \qquad \text{(Str-Comm)}$$

$$((\mathtt{e}_1 \mid \mathtt{e}_2) \mid \mathtt{e}_3) \equiv (\mathtt{e}_1 \mid (\mathtt{e}_2 \mid \mathtt{e}_3)) \qquad \text{(Str-Assoc)}$$

The third rule, called *scope extrusion* in the π-calculus literature, plays a crucial role in communication.

$$\frac{\mathtt{x} \notin FV(\mathtt{e}_2)}{((\mathtt{new\ x{:}T\ e}_1) \mid \mathtt{e}_2) \equiv (\mathtt{new\ x{:}T\ } (\mathtt{e}_1 \mid \mathtt{e}_2))} \qquad \text{(Str-Extrude)}$$

Informally, it says that the scope of the channel `x`, which starts out private to the process $\mathtt{e}_1$, can be extended to include $\mathtt{e}_2$. The side-condition $\mathtt{x} \notin FV(\mathtt{e}_2)$ ensures that $\mathtt{e}_2$ does not already have a free channel named `x`. (This condition can always be satisifed by α-converting the bound name `x` in the expression $(\mathtt{new\ x{:}T\ e}_1)$ before applying the scope extrusion rule.) For example, the process expression `((new x:T c!x) | e)` may be transformed to `(new x:T (c!x | e))`, if `e` does not have `x` as a free variable. It is this rule that allows the new channel `x` to be communicated outside of its original scope to a process in `e`.

2.7 Substitution and Matching

To define reduction, we need some notation for matching values against patterns.

A *substitution* is a finite map associating variables with values and type variables with types. If σ_1 and σ_2 are substitutions with disjoint domains, then $\sigma_1 \cup \sigma_2$ is a substitution that combines the effects of σ_1 and σ_2. A substitution is extended to a function from values to values by applying it to variables that fall in its domain and leaving the rest unchanged. For example, applying the substitution $\sigma = \{\mathtt{x} \mapsto \mathtt{a}\} \cup \{\mathtt{y} \mapsto \mathtt{[]}\}$ to the value `[z [x] x y]`, written $\sigma($`[z [x] x y]`$)$, yields `[z [a] a []]`. Substitution is extended in the usual way to an operation on processes, renaming bound variables as necessary to avoid capture.

When a value `v` is successfully matched by a pattern `p`, the result is a substitution $\{\mathtt{p} \mapsto \mathtt{v}\}$, defined as below. (If `v` and `p` do not have the same structure, then $\{\mathtt{p} \mapsto \mathtt{v}\}$ is undefined. The typing rules ensure that this cannot happen in well-typed programs.)

$$
\begin{array}{rcl}
\{\mathtt{x:T} \mapsto \mathtt{v}\} & = & \{\mathtt{x} \mapsto \mathtt{v}\} \\
\{\mathtt{\_:T} \mapsto \mathtt{v}\} & = & \{\,\} \\
\{\mathtt{(x:T@p)} \mapsto \mathtt{v}\} & = & \{\mathtt{x} \mapsto \mathtt{v}\} \cup \{\mathtt{p} \mapsto \mathtt{v}\} \\
\{\mathtt{(rec:T\ p)} \mapsto \mathtt{(rec:S\ v)}\} & = & \{\mathtt{p} \mapsto \mathtt{v}\} \\
\{\mathtt{\{X<S\}p} \mapsto \mathtt{\{T\}v}\} & = & \{\mathtt{X} \mapsto \mathtt{T}\} \cup \{\mathtt{p} \mapsto \mathtt{v}\} \\
\{\mathtt{[l_1p_1 \ldots l_np_n]} \mapsto \mathtt{[l_1v_1 \ldots l_nv_n \ldots]}\} & = & \{\mathtt{p_1} \mapsto \mathtt{v_1}\} \cup \cdots \cup \{\mathtt{p}_n \mapsto \mathtt{v}_n\}
\end{array}
$$

The *match* function traverses the structure of the pattern and the value in parallel, yielding bindings when variables are encountered in the pattern. (Note that the variables bound in a pattern are always distinct, so the $\cup$ operations in the definition of *match* are always well defined.)

The match rule for records allows a record pattern to be matched by a record value with extra fields (at the end of the record). For example, the record pattern `[l=y]` matches the record value `[l=33 m=true]`. This gives rise to simple form of record subtyping which is particularly easy to implement (it is common to allow extra fields to be added anywhere in a record, but this significantly complicates the implementation of records, especially in the presence of separate compilation).

2.8 Reduction

The reduction relation $\mathtt{e_1} \rightarrow \mathtt{e_2}$ may be read as "The process $\mathtt{e_1}$ *can evolve* to the process $\mathtt{e_2}$." That is, the semantics is nondeterministic, specifying only what *can* happen as the evaluation of a program proceeds, not what *must* happen. Any particular execution of a Pict program will follow just one of the possible paths.

The most basic rule of reduction is the one specifying what happens when an input prefix meets an output atom:

$$\frac{\{\mathtt{p} \mapsto \mathtt{v}\} \text{ defined}}{\mathtt{(x!v\ |\ x?p\ =\ e)} \rightarrow \{\mathtt{p} \mapsto \mathtt{v}\}\mathtt{(e)}} \qquad \text{(Red-Comm)}$$

In the case when the input expression is replicated, the communication rule is similar, except that the input expression is not consumed by the act of communication.

$$\frac{\{\mathtt{p} \mapsto \mathtt{v}\} \text{ defined}}{\mathtt{(x!v\ |\ x?*p\ =\ e)} \rightarrow (\{\mathtt{p} \mapsto \mathtt{v}\}\mathtt{(e)\ |\ x?*p\ =\ e)}} \qquad \text{(Red-RComm)}$$

The next two rules allow reduction to proceed under declarations and parallel composition:

$$\frac{\mathtt{e_1} \rightarrow \mathtt{e_2}}{\mathtt{(d\ e_1)} \rightarrow \mathtt{(d\ e_2)}} \qquad \text{(Red-Dec)}$$

$$\frac{\mathtt{e_1} \rightarrow \mathtt{e_3}}{\mathtt{(e_1\ |\ e_2)} \rightarrow \mathtt{(e_3\ |\ e_2)}} \qquad \text{(Red-Prl)}$$

The body of an input expression, on the other hand, *cannot* participate in reductions until after the input has been discharged. Reduction of conditional processes is straightforward. The typing rules ensure that the guard in a closed, well-typed conditional is either `true` or `false`.

$$\mathtt{if\ true\ then\ e_1\ else\ e_2} \rightarrow \mathtt{e_1} \qquad \text{(Red-If-T)}$$

$$\mathtt{if\ false\ then\ e_1\ else\ e_2} \rightarrow \mathtt{e_2} \qquad \text{(Red-If-F)}$$

The structural congruence relation captures the distributed nature of reduction. Any two subprocesses at the "top level" of a process expression (i.e. not guarded by any input prefixes) may be brought into proximity by structural manipulations and allowed to interact.

$$\frac{\mathtt{e}_1 \;\equiv\; \mathtt{e}_2 \;\rightarrow\; \mathtt{e}_3 \;\equiv\; \mathtt{e}_4}{\mathtt{e}_1 \;\rightarrow\; \mathtt{e}_4} \qquad \text{(Red-Str)}$$

Note that the reduction rules do not maintain any particular ordering among messages sent along the same channel. For example, in the process `(x!y | x!z | x?w = e)` either the value `y` or the value `z` may be communicated to the process `x?w = e`.

Strictly speaking, the semantics we have given is defined only for closed programs — we have been intentionally informal about the built in channels (such as `print`) which connect a Pict program to its environment. Work is underway on a more refined semantic framework explicitly incorporating interactions with the environment [Sew96].

2.9 Fairness

Even on closed programs, the reduction semantics of the previous section leaves one important issue unaddressed: it characterizes the set of possible behaviors of a process expression, but makes no commitment as to which of these behaviors will actually be observed when the expression is compiled and executed. For example, there is a valid execution of the process `(new x (x![] | x?*[] = x![] | x?[] = a![]))` in which the output `a![]` is never executed. But a compiler that produced this behavior would be unsatisfactory, since it would fail to capture the programmer's intuitive expectation that the actions of subprocesses running in parallel will be interleaved *fairly*, so that the second input on `x` will eventually succeed.

We are unaware of any work formalising fairness for π-calculus, but Costa and Stirling's work on fairness for CCS [CS87] seems likely to be generalisable to the case of π-calculus. Costa and Stirling consider two kinds of fairness: *weak fairness* stipulates that if a process is continuously able to communicate on a channel then it must eventually be allowed to proceed; *strong fairness* insists that any process which is able to communicate on a channel infinitely often, even if not continuously, must eventually proceed. A weak fairness guarantee is sufficient to ensure that the output `a![]` in the above example will eventually be executed, since the input `x?[] = a![]` is continuously enabled. If, however, the process `x?*[] = x![]` is replaced by a process which does some other communication before sending `[]` along `x`, then a strong fairness guarantee would be required to ensure that the output `a![]` is eventually executed, since the input `x?[] = a![]` is not continuously able to communicate.

In practice, it is relatively easy to achieve a fair execution strategy by using FIFO channel queues and a round-robin policy for process scheduling. This guarantees that a process waiting to communicate on a channel will eventually succeed, assuming that enough partners become available. Our experience of writing applications in Pict has been that this execution strategy works well. For example, the FIFO queueing of the lock channel `l` in the reference cell of Section 4.6 ensures that competing `set` and `get` requests are handled fairly; the replicated choice construct of [PT95] exhibits similar good behavior.

3 Type System

The Pict type system has its roots in the theoretical literature on type systems for the π-calculus [Mil91, Gay93, VH93, PS93, Tur96] and for functional languages, among which its most immediate predecessors are Quest [Car91] and Amber [Car86]. The treatment of subtyping and higher-order polymorphism is based on recent work on static type systems for object-oriented languages [Car84, Bru94, CHC90, PT94, HP95, FM94, AC96, etc.] and the λ-calculus $F^{\omega}_{\leq}$ [Car90, Mit90, PS97]. The rules for channel types are taken from Pierce and Sangiorgi's type system for the pure π-calculus [PS93]. An early version of the Pict type system was presented in [PRT93].

Typed process calculi with related goals have been proposed by Nierstrasz [Nie95] and Vasconcelos [Vas94]. Further refinements to the channel typing discipline incorporating notions of linear channel usage have been studied by Honda [Hon93, HY94, Hon96], and more recently by Kobayashi and Yonezawa [KY94] and the present authors in collaboration with Kobayashi [KPT96].

3.1 Channel Types

Most type systems for process calculi and concurrent languages impose the constraint that each channel must be used throughout its lifetime to carry values of a single type. This restriction greatly simplifies the task of type analysis, since the well-typedness of a parallel composition (`e`$_1$ `|` `e`$_2$) is independent of the ordering of interactions between `e`$_1$ and `e`$_2$.

Since computation in Pict is based purely on communication over channels, the basic elements of its type system are the types of channels and of the values that they carry. For example, a process that outputs a value `v` along a channel `c` is well typed if `c` has type `^T` (read "channel carrying `T`") and `v` has type `T`.

3.2 Recursive Types

Like most programming languages, Pict offers the capability to build and manipulate recursive data structures like lists and trees. Such *recursive types* have received considerable attention in the literature [MPS86, CC91, AC93, etc.], and many different technical treatments have been proposed. Because the rest of the Pict type system is already somewhat complex and recursive types tend to be used only in small sections of code, we have chosen one of the simplest alternatives, where the "folding" and "unfolding" of the recursion must be managed explicitly by the programmer.

For example, suppose `R` is the recursive type `(rec X = ^X)`. A value of type `R` can be coerced (by means of a `rec` pattern) to a value of type `^R` (where the recursion in the type `R` has been unfolded once). Dually, a `rec` value construct can be used to coerce a value of type `^R` into a value of `R`.

We can use the recursive type `(rec L = ^[^[] ^[Int L]])` to represent the type of integer lists in our "cons cell server" from Section 2.5:

```
type IntList = (rec L = ^[^[] ^[Int L]])

cons?*[hd:Int tl:IntList r:^IntList] =
  (new l:^[^[] ^[Int IntList]]
   (r!(rec:IntList l) | l?*[n:^[] c:^[Int IntList]] = c![hd tl]))
```

The type annotations on `hd`, `tl` and `r` indicate that `cons` takes as arguments an integer and an integer list, and returns an integer list along the channel `r`. The type of the new channel `l` is an unfolding of the type `IntList`. The unfolded `IntList` type exposes the fact that a list is represented as a channel, and enables us to use `l` in a replicated input operation. However, when we return `l` along the result channel `r`, we coerce the type of `l` to `IntList` (using a `rec` value construct).

Values of recursive type are "unfolded" during communication by patterns of the form `(rec:T p)`. For example, if `c` is a channel of type `^IntList`, then the bound variable `l` in the body of the process `c?(rec:IntList l) = ...` has type `^[^[] ^[Int IntList]]`, the unfolding of `IntList`.

3.3 Subtyping

Channel types serve a useful role in ensuring that all parts of a program use a given channel in a consistent way, eliminating the possibility of pattern matching failure (cf. Section 2.7) at run time. Of course, pattern matching failure is just one kind of bad behavior that programs may exhibit; especially in concurrent programs, the range of possible programming mistakes is vast: there may be unintended deadlocks, race conditions, and protocol violations of all kinds. Ultimately, one might hope to see static analysis tools capable of detecting many of these errors, but the technology required to do this is still far off. Fortunately, there are some simple ways in which channel types can be enriched so as to capture useful properties of programs while remaining within the bounds of current typechecking technology.

In Pict, it is relatively rare for a channel to be used for both input and output in the same region of the program; typically, some parts of a program use a given channel only for reading while in others it is used only for writing. For example, the "cons cell server" in the example above only reads from the channel `cons`, while clients only write to `cons`. Similarly, given a request `[hd tl r]`, the server only writes to the result channel `r`, while the client only reads from it.

Pict exploits this observation by providing two refinements of the channel type ^T: a type !T giving only the capability to write values of type T and, symmetrically, a type ?T giving only the capability to read values of type T. For example, we can refine our type annotations for cons as follows:

```
type IntList = (rec L = ![![] ![Int L]])

cons?*[hd:Int tl:IntList r:!IntList] =
  (new l:^[![] ![Int IntList]]
   (r!(rec:IntList l) | l?*[n:![] c:![Int IntList]] = c![hd tl]))
```

The refined type annotations make it clear that cons only requires write capability on the channels r, n, and c. Note that the channel l is created with both read and write capabilities. The cons cell server uses l's read capability locally and gives the client the write capability.

The types ^T, ?T, and !T fall naturally into a *subtype relation* [PS93], since a channel of type ^T may always be used in a context where one of type ?T or !T is expected (for example, in an input or output expression).

3.4 Record Subtyping

One objective of the Pict project was to explore the applicability of our earlier theoretical work on type systems for object-oriented programming. In [PT94], we proposed a powerful combination of subtyping and polymorphism as a basis for statically-typed object-oriented programming in functional languages; equipping Pict with a similar type system makes it a useful testbed for experiments with statically typed concurrent objects.

We implement a simple form of record subtyping which allows record fields to be added to the end of a record. For example, the record type [l=Int m=Bool] is a subtype of [l=Int]. Unlike some record type systems, the order of the fields in a Pict record is significant. For example, [m=Bool l=Int] is *not* a subtype of [l=Int m=Bool]. Pict's simple record subtyping, in combination with the fact that the order of record fields is significant, simplifies the compilation of record values, since the position of a field in a record can be determined from its type at compile time (this is especially useful when separately compiling Pict programs).

3.5 Polymorphism

Our type system may readily be extended to include polymorphism, just as simply typed λ-calculus can be extended with polymorphism [Gir72, Rey74]. We support polymorphic communications by adding two new syntactic forms: package values {T}v and package patterns {X}p. For example, if c is a channel of type ^Int, the output expression z!{Int}[5 c] sends along the channel z the type Int and the pair of values 5 and c. The type of z itself is ^{X}[X ^X], pronounced "channel carrying a type X, a value belonging to type X, and a channel carrying elements of X." In more familiar notation, this type might be written ^(∃X.[X !X]). A process receiving from z has the form z?{X}[v:X x:^X] = ..., which binds the type variable X to the received type. The bound variables v and x have types X and ^X. This effectively means that the only legal operation on v is to send it along x.

We can now generalise our cons cell server so that it is polymorphic in the list element type:

```
cons ?* {X}[hd:X tl:(List X) r:!(List X)] =
  (new l:^[![] ![X (List X)]]
   (r!(rec:(List X) l) | l?*[n:![] c:![X (List X)]] = c![hd tl]))
```

Clients of the polymorphic cons must now send an additional type argument along the cons channel. For example, the following process uses the polymorphic cons to build an integer cons cell (we assume the tail of the list, tl, has already been built and has type (List Int)):

```
(new r:^(List Int) (cons!{Int}[33 tl r] | r?l:(List Int) = ...))
```

Polymorphism and subtyping are combined by giving each bound type variable in a package value an *upper bound*, as in the polymorphic λ-calculus with bounded quantification, System $F_{\leq}$ [CW85, CMMS94].

For example, the type `^{X<T}[X ^X]` describes a channel that can be used to transmit a type `X` and two values of types `X` and `^X`, but also stipulates that the only legal values of `X` are subtypes of `T`.

Just as functions in Pict have no special status — being regarded as output channels on which clients can send tuples of arguments and a continuation channel where the function is to send the result — polymorphic functions are represented as output channels carrying package values. This "pun" entails that the primitive form of polymorphism in Pict is existential types, not universal types as in most typed λ-calculi.

3.6 Type Operators

Strictly speaking, the type `(List Int)` is formed by applying the type constructor `List` to the type `Int`. That is, `List` is a function from types to types. To avoid nonsensical applications like `(List List)` or `(Int Int)`, we classify types and type operators according to their *kinds*, as in typed λ-calculi such as System F^{ω} [Gir72, Bar92] and $F^{\omega}_{\leq}$ [Car90, Mit90, HP95, PS97]. Thus, the type system recognizes three distinct levels of expressions: values, types, and kinds. The level of values contains familiar entities like `5`, `true`, the tuple `[5 true]`, and channels. The level of types contains *proper types* like `Int`, `Bool`, `(List Int)`, `[Int Bool]`, and `^Int`, as well as *type operators* like `List`. The proper types classify values, in the sense that entities at the level of values may inhabit proper types: `5` inhabits `Int`, etc. In the same sense, kinds classify types: all the proper types inhabit the kind `Type`; type operators accepting one proper type parameter and yielding a proper type (like `List`) inhabit the kind `(Type->Type)`; type operators taking two proper type arguments and yielding a proper type inhabit the kind `(Type->(Type->Type))`; and so on.

3.7 Type Inference

Although Pict's core language is explicitly typed, it is very convenient to allow some type annotations to be omitted from user programs. Some languages, such a ML and Haskell, which are based on the Hindley-Milner type system, can automatically infer all necessary type annotations. Pict's type system, however, is significantly more powerful than the Hindley-Milner type system (since, in particular, it allows higher-order polymorphism and subtyping). This unfortunately means that we don't have an algorithm which can infer all the necessary type annotations in a Pict program. Instead, we use a simple *partial* type inference algorithm (the algorithm is partial, in the sense that it may sometimes have to ask the user to add more explicit type information rather than determine the types itself).

Pict's partial type inference algorithm exploits the fact that there are a number of common cases where the type assigned to a bound variable is completely determined by the surrounding program context. For example, the variable `x` in the input expression `c?x=e` has type `Int` if the channel `c` is known to have type `^Int`. Pict's type inference algorithm is local, in the sense that it only uses the immediately surrounding program context to try and fill in a missing type annotation. This might at first seem rather restrictive, but our experience so far has been very favorable. (Our largest Pict program is approximately 6000 lines long, and there are very few cases where one feels that the type inference algorithm isn't inferring enough type annotations automatically.) One of the reasons partial type inference works well in Pict is that many programs already contain explicit type annotations (for the purposes of documentation). It turns out that in many cases these explicit type annotations are sufficient to uniquely determine the types which should be assigned to all other bound variables.

A simple type inference algorithm has two important benefits. Firstly, it makes it easy for Pict programmers to understand the process of type inference (and thereby understand where type annotations are required and what type errors mean). Secondly, a simple type inference algorithm is easier to formalise: Pict's type inference algorithm forms part of the *specification* of the Pict language. Type systems for languages such as ML and Haskell can be specified by means of a set of typing rules which non-deterministically pick the 'correct' types for all bound variables. No details of the actual process of type inference are required (though it is necessary to prove that a sound and complete type inference algorithm does exist). Since we cannot infer all missing type annotations in Pict programs, it is necessary to specify exactly which type annotations can be inferred automatically. Because of the local nature of partial type inference in Pict, it is possible to describe the algorithm using rules which look much like Pict's typing rules, but which formalise how type information propagates into and out of an expression.

In this paper, we concentrate on the explicitly typed language, and omit details about type inference. See the Pict definition [PT97] for a formal description of type inference.

3.8 Notation

The syntax of type expressions is as follows:

Type	=	`^` *Type*	Input/output channel
		`!` *Type*	Output-only channel
		`?` *Type*	Input-only channel
		`{` *Id* `<` *Type* `}` *Type*	Package type
		`[` *Label Type* ... *Label Type* `]`	Record type
		Id	Type identifier
		`\` *Id* `:` *Kind* `=` *Type*	Type operator
		`(` *Type Type* `)`	Type application
		`(rec` *Id* `:` *Kind* `=` *Type* `)`	Recursive type
		`Top :` *Kind*	Maximal type
		`Int`	Integer type
		`Char`	Character type
		`Bool`	Boolean type
		`String`	String type
Kind	=	`(` *Kind* `->` *Kind* `)`	Kind of type operators
		`Type`	Kind of types

A *typing context* Γ is a list of bindings associating variables with their types and type variables with their upper bounds. The metavariables Γ and Δ range over contexts. The concatenation of Γ and Δ is written Γ, Δ.

The type system of Pict comprises axioms and inferences rules defining sets of derivable *statements* of the following forms:

$\Gamma \vdash \mathtt{S} < \mathtt{T}$	`S` is a subtype of `T`
$\Gamma \vdash \mathtt{v} \in \mathtt{T}$	value `v` has type `T` under assumptions Γ
$\Gamma \vdash \mathtt{d} \rhd \Delta$	declaration `d` is well formed and yields bindings Δ
$\Gamma \vdash \mathtt{p} \in \mathtt{T} \rhd \Delta$	pattern `p` requires type `T` and yields bindings Δ
$\Gamma \vdash \mathtt{a} \in \mathtt{T}$	abstraction `a` is well formed and accepts type `T`
$\Gamma \vdash \mathtt{e}\ ok$	process expression `e` is well formed
$\Gamma \vdash \mathtt{T} \in \mathtt{K}$	type `T` has kind `K`
$\vdash \Gamma\ ok$	context Γ is well formed

The first two kinds of statement are familiar from type systems for functional languages. The third is used for checking Pict declarations. Since a declaration cannot be sent over a channel, it does not itself have a type; however, it may give rise to a collection of variable bindings for some following scope, and we need to keep track of the types of these variables. The "type" of a declaration is therefore a typing context. Similarly, a pattern binds some variables and thus gives rise to a context; however, a pattern *also* has a type, since it can only match values of a certain form. An abstraction requires an argument of a certain form. A process expression yields neither bindings nor a value; it is simply either well formed or not. (A process is well formed in a given context if all its input- and output-subexpressions respect the typings of the channels over which communication occurs.) The last two forms of statements give standard rules for well-formedness of types and typing contexts.

The rules for well-kinded types and well-formed contexts are familiar from the literature on higher-order typed λ-calculi (e.g. [HP95, PS97]), and we do not discuss them here. The rest of this section presents a selection of the rules which define the remaining forms of typing statements. (A full description of the typing and kinding rules can be found in the Pict language definition [PT97].)

3.9 Subtyping

The subtype relation consists of two structural rules plus one or more rules for each type constructor or constant. The structural rules state that subtyping is reflexive and transitive and includes β-conversion on types so that, if `F` is the type operator `\X:Type = [X X]`, then `(F Int)` is equivalent to `[Int Int]`:

$$\frac{\mathtt{S} =_{\beta T} \mathtt{T}}{\Gamma \vdash \mathtt{S} < \mathtt{T}} \qquad \text{(S-Conv)}$$

$$\frac{\Gamma \vdash \mathtt{S} \in \mathtt{K} \qquad \Gamma \vdash \mathtt{U} \in \mathtt{K} \qquad \Gamma \vdash \mathtt{T} \in \mathtt{K} \qquad \Gamma \vdash \mathtt{S} < \mathtt{U} \qquad \Gamma \vdash \mathtt{U} < \mathtt{T}}{\Gamma \vdash \mathtt{S} < \mathtt{T}} \qquad \text{(S-Trans)}$$

Formally, the conversion relation $=_{\beta T}$ contains both ordinary β-conversion ($(\mathtt{\backslash X{:}K{=}T)\ S} =_{\beta T} \{\mathtt{X} \mapsto \mathtt{S}\}\mathtt{T}$) and a rule of "top-conversion" (cf. [PS97]) that makes `Top` at operator kinds behave like a type operator ($\mathtt{Top{:}(K_1\text{->}K_2)\ S} =_{\beta T} \mathtt{Top{:}K_2}$).

Each type variable is a subtype of the upper bound declared for it in the context:

$$\Gamma_1, \mathtt{X<T}, \Gamma_2 \vdash \mathtt{X} < \mathtt{T} \qquad \text{(S-TVar)}$$

`Top:K` is a maximal type for each kind `K`. In particular, `Top:Type`, which may be written just `Top`, is the largest type.

$$\Gamma \vdash \mathtt{S} < \mathtt{Top{:}K} \qquad \text{(S-Top)}$$

A record type `S` is a subtype of another record type `T` whenever `S` contains more fields than `T` (and where the types of the corresponding field values are also subtypes). For example, if `Char` < `Int` then `[l=Char m=Bool]` < `[l=Int]`.

$$\frac{\Gamma \vdash \mathtt{T}_1 < \mathtt{T}'_1 \quad \ldots \quad \Gamma \vdash \mathtt{T}_n < \mathtt{T}'_n}{\Gamma \vdash \mathtt{[l_1T_1 \ldots l_nT_n \ldots]} < \mathtt{[l_1T'_1 \ldots l_nT'_n]}} \qquad \text{(S-Record)}$$

The package type $\mathtt{\{X<S_1\}S_2}$ is a subtype of $\mathtt{\{X<T_1\}T_2}$ if the bounds $\mathtt{S_1}$ and $\mathtt{T_1}$ have the same kind, $\mathtt{S_1}$ is a subtype of $\mathtt{T_1}$, and $\mathtt{S_2}$ is a subtype of $\mathtt{T_2}$ under the assumption that `X` is a subtype of $\mathtt{S_1}$:

$$\frac{\Gamma \vdash \mathtt{S}_1 \in \mathtt{K} \qquad \Gamma \vdash \mathtt{T}_1 \in \mathtt{K} \qquad \Gamma \vdash \mathtt{S}_1 < \mathtt{T}_1 \qquad \Gamma, \mathtt{X<S_1} \vdash \mathtt{S}_2 < \mathtt{T}_2}{\Gamma \vdash \mathtt{\{X<S_1\}S_2} < \mathtt{\{X<T_1\}T_2}} \qquad \text{(S-Package)}$$

The channel constructor `?` is covariant in its argument and `!` is contravariant. Operationally, this captures the observation that, for example, if a given channel `x` is being used in a given context only to read elements of type `T`, then it is safe to replace `x` by another channel `y` carrying elements of type `S`, as long as any element that is read from `y` may safely be regarded as an element of `T` — that is, as long as `S` is a subtype of `T`.

$$\frac{\Gamma \vdash \mathtt{S} < \mathtt{T}}{\Gamma \vdash \mathtt{?S} < \mathtt{?T}} \qquad \text{(S-IChan)}$$

$$\frac{\Gamma \vdash \mathtt{T} < \mathtt{S}}{\Gamma \vdash \mathtt{!S} < \mathtt{!T}} \qquad \text{(S-OChan)}$$

Notice that the contravariance of `!` gives rise to the usual rule of subtyping between types of functions. A function $\mathtt{f} \in \mathtt{S_1} \to \mathtt{S_2}$ is implemented in Pict as a server process reading requests of type $\mathtt{[S_1\ \ !S_2]}$ from a channel `f`, performing the appropriate calculation, and returning its result on the channel provided as its second argument. From the point of view of a caller, the request channel `f` has type $\mathtt{![S_1\ \ !S_2]}$; this type is contravariant in $\mathtt{S_1}$ and covariant in $\mathtt{S_2}$, as expected:

$$\frac{\Gamma \vdash \mathtt{T}_1 < \mathtt{S}_1 \qquad \Gamma \vdash \mathtt{S}_2 < \mathtt{T}_2}{\Gamma \vdash \mathtt{![S_1\ \ !S_2]} < \mathtt{![T_1\ \ !T_2]}}$$

The constructor `^` is invariant in the subtype relation (i.e. `^S` is a subtype of `^T` only when `S` and `T` are equivalent). The type `^T` is a subtype of both `?T` and `!T`. That is, we are allowed to forget either the capability to write or the capability to read on a channel: a channel that can be used for both input and output may be used in a context where just one capability is needed.

$$\Gamma \vdash \mathtt{\hat{}T} < \mathtt{?T} \qquad \text{(S-ChanIChan)}$$

$$\Gamma \vdash \mathtt{\hat{}T} < \mathtt{!T} \qquad \text{(S-ChanOChan)}$$

The subtype relation is extended pointwise from proper types to other kinds: if F and G are type operators, then we say $\mathtt{F} < \mathtt{G}$ if $\mathtt{(F\ T)} < \mathtt{(G\ T)}$ for all appropriately kinded argument types T.

$$\frac{\Gamma, \mathtt{X{<}Top{:}K} \vdash \mathtt{S} < \mathtt{T}}{\Gamma \vdash \mathtt{\backslash X{:}K{=}S} < \mathtt{\backslash X{:}K{=}T}} \qquad \text{(S-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{S} < \mathtt{T}}{\Gamma \vdash \mathtt{(S\ U)} < \mathtt{(T\ U)}} \qquad \text{(S-APP)}$$

For subtyping of recursive types, we use the familiar "Amber rule" [Car86, AC93], which states that (rec X=S) is a subtype of (rec Y=T) if we can show $\mathtt{S} < \mathtt{T}$ under the assumption $\mathtt{X} < \mathtt{Y}$.

$$\frac{\Gamma,\ \mathtt{Y{<}Top{:}K},\ \mathtt{X{<}Y} \vdash \mathtt{S} < \mathtt{T}}{\Gamma \vdash \mathtt{(rec\ X{:}K{=}S)} < \mathtt{(rec\ Y{:}K{=}T)}} \qquad \text{(S-REC)}$$

3.10 Values

If the current context contains the binding x:T for the variable x, then the type of x is T in this context (all bound variables are assumed to be unique, so there is no ambiguity in this rule):

$$\Gamma_1, \mathtt{x{:}T}, \Gamma_2 \vdash \mathtt{x} \in \mathtt{T} \qquad \text{(V-VAR)}$$

If the values $\mathtt{v}_1$ through $\mathtt{v}_n$ have the types $\mathtt{T}_1$ through $\mathtt{T}_n$, then the record value $\mathtt{[l_1v_1\ldots l_nv_n]}$ has the record type $\mathtt{[l_1T_1\ldots l_nT_n]}$.

$$\frac{\Gamma \vdash \mathtt{v}_1 \in \mathtt{T}_1 \quad \ldots \quad \Gamma \vdash \mathtt{v}_n \in \mathtt{T}_n}{\Gamma \vdash \mathtt{[l_1v_1\ldots l_nv_n]} \in \mathtt{[l_1T_1\ldots l_nT_n]}} \qquad \text{(V-RECORD)}$$

A value v can be incorporated into an existential package of type {X<U}T if the "witness type" S is a subtype of U. The actual type of the value v must match the type T after the substitution of S for X.

$$\frac{\Gamma \vdash \mathtt{S} \in \mathtt{K} \qquad \Gamma \vdash \mathtt{U} \in \mathtt{K} \qquad \Gamma \vdash \mathtt{S} < \mathtt{U} \qquad \Gamma \vdash \mathtt{v} \in \{\mathtt{X} \mapsto \mathtt{S}\}\mathtt{T}}{\Gamma \vdash \mathtt{\{S\}v} \in \mathtt{\{X{<}U\}T}} \qquad \text{(V-PACKAGE)}$$

For example, if res has type ^Bool, then the value {Bool}[false res] has type {X}[X ^X], since false has type $\{\mathtt{X} \mapsto \mathtt{Bool}\}\mathtt{X} = \mathtt{Bool}$ and res has type $\{\mathtt{X} \mapsto \mathtt{Bool}\}\mathtt{\hat{}X} = \mathtt{\hat{}Bool}$. Readers familiar with typed λ-calculi will recognize the similarity of this rule to the standard introduction rule for existential types (e.g. [CW85, MP88]). The pattern typing rule P-PACKAGE in Section 3.12 plays the role of the standard elimination rule for existentials.

A value of a recursive type T can be formed from a value of whose type matches the "unrolling" of T:

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type} \qquad \mathtt{T} \leadsto \mathtt{U} \qquad \Gamma \vdash \mathtt{v} \in \mathtt{U}}{\Gamma \vdash \mathtt{(rec{:}T\ v)} \in \mathtt{T}} \qquad \text{(V-REC)}$$

where $\mathtt{T} \leadsto \mathtt{U}$ means that T is a recursive type and U is obtained from T by unrolling the recursion one step. For example, if R is the type (rec X:Type = ^X) and c is a channel of type ^R, then (rec:R c) has type R, since $\mathtt{R} \leadsto \mathtt{\hat{}R}$.

$$\frac{\mathtt{S} =_{\beta T} \mathtt{(rec\ X{:}K\ =\ T)}}{\mathtt{S} \leadsto \{\mathtt{X} \mapsto \mathtt{(rec\ X{:}K\ =\ T)}\}\mathtt{T}} \qquad \text{(UNROLL-REC)}$$

In general, the unrolling operator $\leadsto$ must take into account the fact that the unrolling operation may be applied to a type expression formed by applying a recursively defined type operator to some arguments; in this case, the arguments are carried along unchanged to the result and the recursive type is unrolled "in-place":

$$\frac{\mathtt{S} =_{\beta T} \mathtt{(T_1\ T_2)} \qquad \mathtt{T_1} \leadsto \mathtt{U}}{\mathtt{S} \leadsto \mathtt{(U\ T_2)}} \qquad \text{(UNROLL-APP)}$$

Finally, we allow types of values to be *promoted* in the subtype relation: if v is a value of type S and S is a subtype of T, then v also has type T.

$$\frac{\Gamma \vdash \mathtt{v} \in \mathtt{S} \qquad \Gamma \vdash \mathtt{S} < \mathtt{T}}{\Gamma \vdash \mathtt{v} \in \mathtt{T}} \qquad \text{(V-SUB)}$$

This rule embodies the principle of "safe substitutability" that underlies the subtype relation: the statement $\mathtt{S} < \mathtt{T}$ means that an element of S can always be used in a context where an element of T is required.

3.11 Declarations

A `new` declaration returns a binding for the new channel using the declared type (we check that the declared type is well-kinded and equivalent to a channel type).

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type} \qquad \Gamma \vdash \mathtt{T} =_{\beta T} \mathtt{\hat{}U}}{\Gamma \vdash \mathtt{new\ x{:}T} \triangleright \mathtt{x{:}T}} \tag{D-New}$$

3.12 Patterns

Pattern typing statements have the form $\Gamma \vdash \mathtt{p} \in \mathtt{T} \triangleright \Delta$. That is, each pattern has a type, describing the shape of the values that it can match, and moreover gives rise to a set of type- and term-variable bindings.

A variable pattern `x:T` matches any value of type `T` and gives rise to a binding for the variable `x`.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type}}{\Gamma \vdash \mathtt{x{:}T} \in \mathtt{T} \triangleright \mathtt{x{:}T}} \tag{P-Var}$$

A wildcard pattern `_:T` matches any value of type `T` but does not give rise to any variable bindings.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type}}{\Gamma \vdash \mathtt{\_{:}T} \in \mathtt{T} \triangleright \bullet} \tag{P-Wild}$$

A layered pattern `x:T@p` matches a value of type `T`. We return whatever variables are bound in `p`, plus a binding for `x`.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type} \qquad \Gamma \vdash \mathtt{p} \in \mathtt{T} \triangleright \Delta}{\Gamma \vdash \mathtt{x{:}T@p} \in \mathtt{T} \triangleright \mathtt{x{:}T}, \Delta} \tag{P-Layered}$$

A `rec` pattern accepts a value of type `T`, but the subpattern `p` is matched against a value with the unfolded recursive type `U`.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type} \qquad \mathtt{T} \rightsquigarrow \mathtt{U} \qquad \Gamma \vdash \mathtt{p} \in \mathtt{U} \triangleright \Delta}{\Gamma \vdash \mathtt{(rec{:}T\ p)} \in \mathtt{T} \triangleright \Delta} \tag{P-Rec}$$

A record pattern $\mathtt{[l_1p_1 \ldots l_np_n]}$ has the type $\mathtt{[l_1T_1 \ldots l_nT_n]}$, where the $\mathtt{T}_i$'s are the types of its elements, and gives rise to a set of bindings including all the bindings from its subpatterns.

$$\frac{\Gamma \vdash \mathtt{p}_1 \in \mathtt{T}_1 \triangleright \Delta_1 \quad \ldots \quad \Gamma \vdash \mathtt{p}_n \in \mathtt{T}_n \triangleright \Delta_n}{\Gamma \vdash \mathtt{[l_1p_1 \ldots l_np_n]} \in \mathtt{[l_1T_1 \ldots l_nT_n]} \triangleright \Delta_1, \ldots, \Delta_n} \tag{P-Record}$$

A package pattern `{X<U}p` matches any value of type `{X<U}T`, where `T` is the type of the pattern `p` (under the assumption that `X` is a subtype of `U`). The pattern `{X<U}p` yields not only the bindings produced by `p`, but also the type binding `X<U`.

$$\frac{\Gamma \vdash \mathtt{U} \in \mathtt{K} \qquad \Gamma, \mathtt{X{<}U} \vdash \mathtt{p} \in \mathtt{T} \triangleright \Delta}{\Gamma \vdash \mathtt{\{X{<}U\}p} \in \mathtt{\{X{<}U\}T} \triangleright \mathtt{X{<}U}, \Delta} \tag{P-Package}$$

3.13 Process Abstractions

A process abstraction `p=e` requires an argument of type `T`, where `T` is the type of the pattern `p`. The process `e` is typechecked in a context extended with the bindings Δ introduced by `p`.

$$\frac{\Gamma \vdash \mathtt{p} \in \mathtt{T} \triangleright \Delta \qquad \Gamma, \Delta \vdash \mathtt{e}\ \mathit{ok}}{\Gamma \vdash \mathtt{p{=}e} \in \mathtt{T}} \tag{A-Abs}$$

3.14 Processes

The typing rules for processes are the simplest of all. The parallel composition of two processes is well formed in a given context if both parts are.

$$\frac{\Gamma \vdash \mathtt{e}_1\ \mathit{ok} \qquad \Gamma \vdash \mathtt{e}_2\ \mathit{ok}}{\Gamma \vdash \mathtt{(e_1\ \mid\ e_2)}\ \mathit{ok}} \tag{E-Prl}$$

An input expression `v?a` is well formed if `v` is a channel for which we have input permission, i.e., it has type `?T` for some `T`, and `a` is a well formed abstraction which accepts a value of type `T`.

$$\frac{\Gamma \vdash \mathtt{v} \in \mathtt{?T} \qquad \Gamma \vdash \mathtt{a} \in \mathtt{T}}{\Gamma \vdash \mathtt{v?a}\ ok} \tag{E-In}$$

Symmetrically, an output expression $\mathtt{v_1!v_2}$ is well formed if $\mathtt{v_1}$ has an output channel type $\mathtt{!T}$, for some $\mathtt{T}$, and $\mathtt{v_2}$ has type $\mathtt{T}$.

$$\frac{\Gamma \vdash \mathtt{v_1} \in \mathtt{!T} \qquad \Gamma \vdash \mathtt{v_2} \in \mathtt{T}}{\Gamma \vdash \mathtt{v_1!v_2}\ ok} \tag{E-Out}$$

Finally, a local declaration provides a set of bindings Δ in which the process body is checked.

$$\frac{\Gamma \vdash \mathtt{d} \rhd \Delta \qquad \Gamma, \Delta \vdash \mathtt{e}\ ok}{\Gamma \vdash \mathtt{(d\ e)}\ ok} \tag{E-Dec}$$

A conditional expression is well formed if the guard expression has boolean type, and the two branches of the conditional are well formed.

$$\frac{\Gamma \vdash \mathtt{b} \in \mathtt{Bool} \qquad \Gamma \vdash \mathtt{e_1}\ ok \qquad \Gamma \vdash \mathtt{e_2}\ ok}{\Gamma \vdash \mathtt{if\ b\ then\ e_1\ else\ e_2}\ ok} \tag{E-If}$$

3.15 Type Safety

The relation between the type system and the operational semantics can be expressed in the form of two slogans: *evaluation cannot fail in well-typed processes*, and *reduction preserves typing.* We define runtime failure by means of a set of inference rule similar in form to Pict's reduction rules (for the sake of brevity, we only present the most important rules).

The most important type of failure we hope to prevent is pattern-matching failure during communication (this type of failure can also occur in a communication with a replicated input, but we omit that rule here):

$$\frac{\{\mathtt{p} \mapsto \mathtt{v}\}\ \text{undefined}}{\mathtt{(x!v\ \mid\ x?p\ =\ e)}\ fails} \tag{Fail-Comm}$$

In addition, a process fails if it attempts to use any value other than a channel as the subject of a communication (recall that we do not have any syntax for channel constants, only variables which range over channels). For example, `[]!23` *fails*, since it attempts to use the record value `[]` as a channel. (We omit the rules for similar failures in input and replicated input prefixes.)

$$\frac{\mathtt{v_1}\ \text{is not a variable}}{\mathtt{v_1!v_2}\ fails} \tag{Fail-Out}$$

Failures may also occur inside local declarations and parallel compositions of processes (the Fail-Str rule, in combination with Fail-Prl, captures the case when a failure occurs in the right-hand subterm of a parallel composition):

$$\frac{\mathtt{e}\ fails}{\mathtt{(d\ e)}\ fails} \tag{Fail-Dec}$$

$$\frac{\mathtt{e_1}\ fails}{\mathtt{(e_1\ \mid\ e_2)}\ fails} \tag{Fail-Prl}$$

We reuse the structural congruence relation (from Section 2.6) to capture the distributed nature of failures. A process is considered to have failed if any two subprocesses at the "top level" (i.e. not guarded by any input prefixes) may be brought into proximity by structural manipulations so that they fail.

$$\frac{\mathtt{e_1} \equiv \mathtt{e_2} \qquad \mathtt{e_2}\ fails}{\mathtt{e_1}\ fails} \tag{Fail-Str}$$

3.15.1 Conjecture [Type safety]: If $\Gamma \vdash \mathtt{e}$ then $\mathtt{e}$ does not fail.

3.15.2 Conjecture [Subject reduction]: If $\Gamma \vdash \mathtt{e_1}$ and $\mathtt{e_1} \rightarrow \mathtt{e_2}$ then $\Gamma \vdash \mathtt{e_2}$.

The metatheoretic foundations needed to prove these two properties have already been established for the major components of the Pict type system — for channel types and subtyping by Pierce and Sangiorgi [PS93], for polymorphic channels by Turner [Tur96], for higher-order polymorphism with subtyping by Pierce and Steffen [PS97] and Compagnoni [Com94]. However, the above properties remain conjectures, since we have not checked the type system as a whole.

4 Derived Forms

The statically-typed core language of Pict is a powerful, safe, and unacceptably verbose programming notation. In this section, we show how more convenient high-level constructs are built up from the core by means of source-to-source translations — following in the tradition of numerous papers showing how various high-level features can be encoded in the π-calculus [San92, San94, San93, Mil90, Jon93, Wal95, Ama94, AP94, etc.]. We discuss only the more interesting translation rules; the complete list can be found in the Pict language definition [PT97].

4.1 Simple Translations

Large programs often contain long sequences of declarations like `(new` $\mathtt{x}_1$ `... (new` $\mathtt{x}_n$ `e))`. To avoid proliferation of parentheses, we introduce the more compact syntactic form `(new` $\mathtt{x}_1$ `... new` $\mathtt{x}_n$ `e)` in the high-level language. Formally, we extend the syntactic category of processes with n-ary declarations of the form `(`$\mathtt{d}_1$ `...` $\mathtt{d}_n$ `e)` and introduce a translation rule

$$(\mathtt{d}_1 \ldots \mathtt{d}_n\ \mathtt{e}) \Rightarrow (\mathtt{d}_1\ \ldots\ (\mathtt{d}_n\ \mathtt{e})) \qquad (\textsc{Tr-DecSeq})$$

that shows how n-ary declarations may be interpreted as expressions in the core language.

In sequences of declarations, it is often convenient to start some process running in parallel with the evaluation of the remainder of the declaration. We introduce the declaration keyword `run` for this purpose. After a declaration sequence has been translated into a nested collection of individual declarations, `run` declarations may be translated into simple parallel compositions:

$$(\mathtt{run}\ \mathtt{e}_1\ \mathtt{e}_2) \Rightarrow (\mathtt{e}_1\ \mid\ \mathtt{e}_2) \qquad (\textsc{Tr-Run})$$

For example, the process

```
(run print!"twittering"
 run print!"rising"
 print!"overhead passing")
```

is transformed by Tr-DecSeq followed by two applications of Tr-Run into:

```
(print!"twittering" | (print!"rising" | print!"overhead passing"))
```

Many variants of the π-calculus allow *process abstractions* like $F(x, y) = \bar{x}y \mid \bar{x}y$. In Pict, such abstractions are introduced via the declaration keyword `def`, as in `def f [x y] = (x!y | x!y)`, and instances are created using the same syntax as output expressions, as in `f![a b]`. The coincidence between the notations for sending on a channel and instantiating a process abstraction is not accidental: we translate a process abstraction like the one above into a channel declaration `new f` and a replicated receiver `f?*[x y] = (x!y | x!y)`, so that instantiating an abstraction actually *is* just an output. Formally, this translation is captured by the following rule:

$$(\mathtt{def\ x\ p\ =\ e}_1\quad \mathtt{e}_2) \Rightarrow (\mathtt{new\ x\ (x?{*}p\ =\ e}_1\ \mid\ \mathtt{e}_2))$$

Recursive and mutually recursive definitions are also allowed. The first definition in a recursive group is introduced by `def`, the others with `and`.

```
def f [x y] = ... g![a b] ...
and g [z w] = ... f![a b] ...
```

The general translation rule, then, is:

$$\begin{array}{c}(\mathtt{def}\ \mathtt{x}_1\mathtt{a}_1\ \ldots\ \mathtt{and}\ \mathtt{x}_n\mathtt{a}_n\ \mathtt{e}) \Rightarrow \\ (\mathtt{new}\ \mathtt{x}_1\ \ldots\ (\mathtt{new}\ \mathtt{x}_n\ (\mathtt{x}_1?{*}\mathtt{a}_1\ \mid\ \ldots\ \mid\ \mathtt{x}_n?{*}\mathtt{a}_n\ \mid\ \mathtt{e}))\ \ldots\ )\end{array} \qquad (\textsc{Tr-Def})$$

Note that Tr-Def is a transformation on typed expressions. However, since the actual type of the channel $\mathtt{x}_i$ is determined by the type of the pattern $\mathtt{p}_i$, we omit the type annotation.

4.2 Complex Values

So far, all the value expressions we have encountered have been built up in an extremely simple way, using just variables, channels, basic values, tuples of values, and records of values. These *simple values* are important because they are exactly the entities that can be passed along channels and participate in pattern matching.

In real programs, it is very common to write an expression that computes a simple value and immediately sends it along some channel. For example, the process `(new n c!n)` creates a fresh channel `n` and sends it off along `c`. An alternative syntax for such expressions, which can often make them easier to understand, puts the whole value-expression *inside* the output: `c!(new n n)`. In general, it is useful to allow such expressions in any position where a simple value is expected. Formally, we extend the syntactic category of values with declaration values of the form `(d v)`. We use the term *complex value* for an expression in the extended syntax that does not fall within the core language.

When we write `c!(new n n)`, we do not mean to send the *expression* `(new n n)` along `c`. A complex value is always evaluated "strictly" to yield a simple value, which is substituted for the complex expression.

In introducing complex values, we have taken a fairly serious step: we must now define the meaning of a complex value occurring in any position where simple values were formerly allowed. For example, the nested expression `c![23 (new x x) (new y y)]` must be interpreted as a core language expression that creates two new channels, packages them into a simple tuple along with the integer `23` and sends the result along `c`.

We interpret arbitrary complex values using a general "continuation-passing" translation. Given a complex value `v` and a continuation channel `c`, $[\![\mathtt{v} \to \mathtt{c}]\!]$ will denote a process that evaluates `v` and sends the resulting simple value along `c`. We then introduce translation rules for process expressions containing complex values. For example, the rule

$$\mathtt{v_1!v_2} \;\Rightarrow\; \mathtt{(new\ c\ (}[\![\mathtt{v_1} \to \mathtt{c}]\!]\ \mathtt{|\ c?x\ =\ }[\![\mathtt{v_2} \to \mathtt{x}]\!]\mathtt{))} \qquad (\textsc{Tr-Out})$$

translates an output $\mathtt{v_1!v_2}$ into a process expression that first allocates a fresh continuation channel `c`, evaluates $\mathtt{v_1}$, waits for its result to be sent along `c`, and then evaluates $\mathtt{v_2}$, sending the result directly along the channel `x` that resulted from the evaluation of $\mathtt{v_1}$. Input processes containing complex values are translated similarly:

$$\mathtt{v?a} \;\Rightarrow\; \mathtt{(new\ c\ (}[\![\mathtt{v} \to \mathtt{c}]\!]\ \mathtt{|\ c?x\ =\ x?a))} \qquad (\textsc{Tr-In})$$

$$\mathtt{v?{*}a} \;\Rightarrow\; \mathtt{(new\ c\ (}[\![\mathtt{v} \to \mathtt{c}]\!]\ \mathtt{|\ c?x\ =\ x?{*}a))} \qquad (\textsc{Tr-RIn})$$

The continuation-passing translation itself is defined by induction on the syntax of value expressions:

$$\begin{array}{rcl}
[\![\mathtt{x} \to \mathtt{c}]\!] & = & \mathtt{c!x} \\
[\![\mathtt{k} \to \mathtt{c}]\!] & = & \mathtt{c!k} \\
[\![\mathtt{(d\ v)} \to \mathtt{c}]\!] & = & \mathtt{(d\ }[\![\mathtt{v} \to \mathtt{c}]\!]\mathtt{)} \\
[\![\mathtt{(rec{:}T\ v)} \to \mathtt{c}]\!] & = & \mathtt{(new\ c'\ (}[\![\mathtt{v} \to \mathtt{c'}]\!]\ \mathtt{|\ c'?x\ =\ c!(rec{:}T\ x)))} \\
[\![\mathtt{\{T\}v} \to \mathtt{c}]\!] & = & \mathtt{(new\ c'\ (}[\![\mathtt{v} \to \mathtt{c'}]\!]\ \mathtt{|\ c'?x\ =\ c!\{T\}x))}
\end{array}$$

Record values are evaluated left-to-right:

$$\begin{array}{rcl}
[\![\mathtt{[l_1v_1 \ldots l_nv_n]} \to \mathtt{c}]\!] & = & \mathtt{(new\ c_1\ (}[\![\mathtt{v_1} \to \mathtt{c_1}]\!]\ \mathtt{|\ c_1?x_1\ =\ \ldots} \\
 & & \quad \mathtt{(new\ c_n\ (}[\![\mathtt{v_n} \to \mathtt{c_n}]\!]\ \mathtt{|\ c_n?x_n\ =} \\
 & & \qquad \mathtt{c![l_1x_1 \ldots l_nx_n]))\ \ldots\ ))}
\end{array}$$

4.3 Value Declarations

Since complex value expressions may become long or involve expensive computations, it is convenient to introduce a new declaration form that evaluates a complex value and names its result. For example, `(val x = v e)` binds `x` to the result of evaluating `v` and then executes `e`. Formally, `val` declarations are translated using the continuation-passing translation:

$$\mathtt{(val\ p{=}v\ e)} \;\Rightarrow\; \mathtt{(new\ c\ (}[\![\mathtt{v} \to \mathtt{c}]\!]\ \mathtt{|\ c?p\ =\ e))} \qquad (\textsc{Tr-Val})$$

Note that when a `val` declaration `(val p=v e)` is translated into the core language, the body `e` appears inside an input prefix. This fact implies that `val` declarations are *strict* or *blocking*: the body cannot proceed until the bindings introduced by the `val` have actually been established.

4.4 Application

Of course, allowing declarations inside values represents only a minor convenience; the usefulness of this extension by itself would not justify all of the foregoing machinery. But having established the basic pattern of simplifying complex value expressions by means of a continuation-passing transformation, we can apply it to a much more useful extension. In value expressions, we allow the *application* syntax $\mathtt{(v\ v_1\ \ldots\ v_n)}$. For example, if we define a `double` function by

```
def double [s:String r:!String] = concat![s s r]
```

(where `concat` is string concatenation), then, in the scope of the declaration, we can write `(double s)` as a value, dropping the explicit result channel `r`. For example, `print!(double "soothe")` causes `"soothesoothe"` to be sent along the built-in channel `print`.

In fact, we allow a slightly more general syntax for application which enables argument values to be labelled and witness types to be provided (in case the operation is polymorphic). We define the meaning of application by adding a clause to the definition of the continuation-passing translation:

$$\begin{aligned}
[\![\mathtt{(v\ |T_1\ldots T_n|\ l_1v_1\ldots l_nv_n)} \to \mathtt{c}]\!] \quad = \quad & \mathtt{(new\ c'\ (}[\![\mathtt{v} \to \mathtt{c'}]\!]\ \mathtt{|\ c'?x\ =\ \ldots} \\
& \quad \mathtt{(new\ c_1\ (}[\![\mathtt{v_1} \to \mathtt{c_1}]\!]\ \mathtt{|\ c_1?x_1\ =\ \ldots} \\
& \quad\quad \mathtt{(new\ c_n\ (}[\![\mathtt{v_n} \to \mathtt{c_n}]\!]\ \mathtt{|\ c_n?x_n\ =} \\
& \quad\quad\quad \mathtt{x!\{T_1\}\ldots\{T_n\}[l_1x_1\ldots l_nx_n\ \ c]))\ldots))))}
\end{aligned}$$

The 'function' value `v` is evaluated first, followed by the argument values $\mathtt{v_1}$ to $\mathtt{v_n}$. Finally, the function is called and instructed to return its result along the application expression's continuation channel `c`.

4.5 Abstractions

Although Pict's core language and type system do not distinguish between "real functions" and processes that act like functions, it is nevertheless often useful to write parts of programs in a functional style. This is supported by a small extension to the syntactic class of abstractions, mirroring the ability to omit the names of result parameters in applications. For example, we replace a process definition of the form $\mathtt{def\ f\ [a_1\ a_2\ a_3\ r]\ =\ r!v}$, where the whole body of the definition consists of just an output of some (complex) value on the result channel `r`, by a "function definition" $\mathtt{def\ f\ (a_1\ a_2\ a_3)\ =\ v}$ that avoids explicitly giving a name to `r`. Formally, this is captured by the following translation rule for abstractions:

$$\begin{gathered}
\mathtt{(|X_1{<}T_1\ldots X_n{<}T_n|\ l_1p_1\ldots l_np_n):T\ =\ v} \Rightarrow \\
\mathtt{\{X_1{<}T_1\}\ldots\{X_n{<}T_n\}[l_1p_1\ldots l_np_n\ \ r:!T]\ =\ r!v}
\end{gathered} \qquad (\text{Tr-VAbs})$$

The derived form also allows for type arguments in a function definition, which are translated to package patterns. Note that the explicit result type annotation `T` becomes a type annotation `!T` on the result channel.

Since anonymous process declarations like `(def x [] = e  x)` or `(def x () = v x)` are frequently useful for higher-order programming, we provide anonymous abstractions as a special form of value. We do not need an extra case in our continuation-passing translation to describe the meaning of this special form: we just add a local transformation on values:

$$\mathtt{\backslash a} \Rightarrow \mathtt{(def\ x\ a\ \ x)} \qquad (\text{Tr-AnonAbs})$$

For example,

```
def applyTwice (f x) = (f (f x))
val y = (applyTwice  \(x) = (+ x 1)  3)
```

defines a function `applyTwice` and passes it an anonymous function that adds one to its argument.

4.6 Examples

To illustrate some of the high-level forms we have introduced, here is the list example from Section 2.5 rewritten using the full syntax:

```
type (List X) = (rec L = ![![] ![X L]])
def nil (|X|) : (List X) = (rec \[n c] = n![])
def cons (|X| hd:X tl:(List X)) : (List X) = (rec \[n c] = c![hd tl])
```

Uses of `nil` and `cons` can also be streamlined by using application syntax:

```
val l = (cons 22 (cons 33 (cons 44 (nil))))
```

The next example illustrates how we can build a simple reference cell object in Pict.

```
def newRef (|X| init:X) =
  (new l:^X
   run l!init
   [set = \[v:X c:![]] = l?x = (l!v | c![])
    get = \[r:!X]      = l?x = (l!x | r!x)])
```

Each message sent along `newRef` consists of a pair of values: `init`, the initial value of the reference cell that is to be created, and an implicit result channel that the server uses to return the newly created reference cell to the requesting client. After reading a request, the server creates a new channel `l` which acts as a "container" for the current value of the reference cell. Sending a value along `l` represents the action of placing a value in the container. Receiving a value from `l` empties the container; it is then the receiver's responsibility to refill the container by transmitting a new value along `l`. The container is initialized by sending `init` along `l`.

In parallel with initializing the container `l`, `newRef` returns a record containing `set` and `get` methods (process abstractions). Each method waits for a request on its service port; having received one, it reads `l` to obtain the current value of the cell, refills `l` as appropriate, and sends a result (or acknowledgement) to the client. It is possible that multiple copies of each method may be running in parallel at any given moment. But since there is never any more than one sender on `l`, all but one of them will be blocked waiting for an input on `l`.

5 Discussion

We now return to the motivating questions from the introduction and summarize what we have learned.

What is it like to program in the π-calculus? What kind of high-level language can be built on it?

The π-calculus is best thought of as a kind of concurrent machine code: it is simple, flexible, and efficiently implementable, and it offers a suitable target for compilation of higher-level language features. Indeed, the variety of features whose semantics can be expressed in terms of message passing is so wide that many quite different language designs could have arisen from our experiment.

It is worth bearing in mind that choosing π-calculus as a semantic framework strongly discourages the use of some potentially useful language features, such as process priorities and exceptions, which cannot be easily formalized in this setting. A particularly important feature that is not addressed by π-calculus is physical distribution, since the semantic framework of the π-calculus lacks necessary concepts such as process location and failure. Work is currently underway on the design of a new language, tentatively named Distributed Pict, based on a variant of the π-calculus [FG96] extended with distribution primitives [FGL$^+$96]. Cardelli's Obliq [Car95] achieves related aims by building on a primitive notion of network objects.

Pict belongs to a sizeable family of concurrent programming language designs inspired by theoretical calculi, including Vasconcelos's TyCo [Vas94], Kobayashi's HACL [Kob96], and numerous actor languages [Hew77, Agh86, etc.]. A particularly close relative is the language Oz [Smo95], which integrates functional, object-oriented, and concurrent constraint programming by translation into a common core calculus [Smo94]. Although this calculus uses concurrent constraints as its basic communication mechanism, the encoding of high-level features is strongly reminiscent of Pict.

Our choice of high-level language features leads to a programming style similar to that found in functional languages with channel-based concurrency such as PFL [Hol83], Amber [Car86], CML [Rep91, BMT92], Facile [GMP89], Poly/ML [Mat91], and Concurrent Haskell [JGF96]. The most significant difference lies in the type system: the impredicative polymorphism of Pict permits the encoding of polymorphic functions using polymorphic communication. This pun is not possible in languages whose type systems are based on ML polymorphism, where channels cannot carry messages of varying types. Also, the refined channel types provided by Pict (such as input-only and output-only channels) give the programmer useful extra control over channel usage in programs. Languages such as CML, Poly/ML, or Concurrent Haskell do not distinguish different modes of channel usage (and therefore also miss the opportunity to optimise the implementation of communication by exploiting explicit type information).

What kinds of concurrent objects arise in this setting?

We have found that a simple style of objects arises almost necessarily in message-based concurrent programming: an object is just a group of agents that cooperate to provide some collection of services to the "outside world," jointly maintaining the consistency of some shared data. It is convenient to group these services together as a record of named channels, allowing access to the whole collection of services to be passed around between clients as a single unit.

Unfortunately, the more subtle mechanisms found in many concurrent object-oriented languages, such as dynamic method lookup and inheritance of synchronization policies, do not arise in the same "inevitable" way. Rather than commit to a particular high-level object model in Pict, we have chosen to provide a framework for experimenting with a variety of designs. Pict's type system incorporates a number of powerful constructs, such as higher-order subtyping, especially for this purpose. Preliminary experiments with concurrent objects in Pict are described in [PT95]. Some more sophisticated proposals are described in [NSL96].

Can the π-calculus be implemented efficiently?

Pict's high-level language is defined by means of a translation into a π-calculus core language. This is a very useful style of definition as far as the compilation of Pict is concerned, since it identifies a very small calculus which is sufficient to implement the whole of Pict. The operational semantics of π-calculus, plus a number well-known program equivalences, give rise a number of easy to implement (and provably correct) program optimisations, many of which generalise optimisations already commonly used in compilers for functional languages. Our Pict compiler does all of its static analysis of programs, optimisation, and code generation using a π-calculus core language.

However, encoding a high-level language into a low-level language such as π-calculus does run the risk of losing useful information about a program. Fortunately, we have so far been able to regain the information we need by exploiting explicit type information (in particular, we make heavy use of type information to optimise the implementation of communication).

Functional code, when compiled by our Pict compiler comes out looking very like the code generating by a continuation-passing compiler. We compile to C for portability and easy inter-operability with existing program libraries, though this does have a significant cost in efficiency for the compiled code (Tarditi, Archarya, and Lee [TAL90] report that when they modified the New Jersey SML compiler so that it generated C code, it produced code which ran approximately twice as slow as code produced by the native code generator).

Very simple comparisons of the code produced by Pict and New Jersey SML [Tur96] indicate that functional code compiled by Pict runs approximately six times slower than that produced by New Jersey SML. We find this quite encouraging, since the Pict compiler has had very little tuning and lacks a number of important optimisations (in particular, the representation of closures in Pict is not yet optimised in any way). Moreover, New Jersey SML has the advantage of compiling to native code (the code we generate is very similar to the code generated by Tarditi, Archarya, and Lee's `sml2c` compiler, so we might reasonably expect to gain a factor of two if we produced native code instead of C code, which would leave us within a factor of three of the performance of New Jersey SML).

To give an idea of how fast our channel-based communication primitives are, we compared the performance of the Pict `nqueens` program with an equivalent CML program which uses CML's channel primitives to implement the result channels used in Pict [Tur96]. The CML program ran almost four times slower than Pict. This is not to say that CML programs in general run four times slower than Pict, since CML programs typically consist of large amounts of SML code, which runs faster than Pict. However, the comparison does give an idea of the raw performance of Pict's communication primitives. (Especially since the CML program had the advantage of being compiled to native code.)

Can we design a practical type system for the π-calculus combining subtyping and higher-order polymorphism?

The Pict type system integrates a number of well-studied ideas: Milner's simple sorting discipline for channels [Mil91], polymorphic channels [Tur96], higher-order polymorphism [Gir72], input/output modalities [PS93], higher-order subtyping [Car90, Mit90, PT94, HP95, PS97], and recursive types [MPS86, AC93]. However, the key to obtaining a workable type system for Pict was the development of a practical type inference algorithm. Pict's partial type inference algorithm is surprisingly simple and easy to understand, but yet our experience has been that it gives very acceptable results. One of the reasons partial type inference works well in Pict is that many programs already contain explicit type annotations (for the purposes of documentation). It turns out that in many cases these explicit type annotations are sufficient to uniquely determine the types which should be assigned to all other bound variables.

In the design of Pict's type system we gave up the goal of complete type inference in preference for more powerful type-theoretic constructs. For example, Pict's impredicative polymorphism directly supports useful features such as first-class existential types, which are not expressible in simpler, predictive, polymorphic type systems. Thus, without any further extensions to the language, Pict programmers can structure programs using abstract datatypes (this facility is used extensively throughout Pict's standard libraries). We are working on modest extensions to Pict's type system which will enable better 'programming in the large' but, unlike Standard ML, will not require a separate module-level language. For instance, we hope to extend Pict's treatment of existential types to account for type sharing (using techniques similar to those proposed by Leroy [Ler95] and Harper and Lillibridge [HL94]).

Acknowledgements

Robin Milner's past and present work on programming languages, concurrency, and the π-calculus in particular is very strongly in the background of this project, and conversations with Robin have contributed many specific insights. The idea of basing a programming language design on the π-calculus was planted by Bob Harper and developed into a research project in the summer of 1992 in discussions of concurrent object-oriented programming languages with the Edinburgh ML Club. From Davide Sangiorgi, we learned about the higher-order π-calculus and the many ways of encoding λ-calculi in the π-calculus. Didier Rémy helped build the original PIC compiler (on which the first version of the present Pict compiler was based [PRT93]) and joined in many discussions about the integration of processes and functions. Uwe Nestmann's research on proof techniques for compilations between concurrent calculi sharpened our ideas about the formal foundations of Pict. Martin Steffen helped study the formal foundations of the subtyping algorithm. Dilip Sequeira contributed both code and ideas to the implementation of type inference and record type checking. Kohei Honda, Nobuko Yoshida, and Peter Sewell helped us think about fairness in process calculi. Conversations with Luca Cardelli, Georges Gonthier, Sigbjorn Finne, Cliff Jones, Naoki Kobayashi, Martin Müller, Joachim Niehren, Oscar Nierstrasz, Simon Peyton Jones, John Reppy, Gert Smolka, and David Walker have deepened our understanding of the π-calculus and concurrent programming languages.

Pierce has been supported by fellowships from the U.K. Science and Engineering Research Council and Engineering and Physical Sciences Research Council and by the ESPRIT Basic Research Actions TYPES and CONFER. Turner has been supported by Harlequin Ltd. and a fellowship from the U.K. Engineering and Physical Sciences Research Council.

References

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. A preliminary version appeared in POPL '91 (pp. 104–118) and as DEC Systems Research Center Research Report number 62, August 1990.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[ACS96] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous pi-calculus. Technical report, INRIA-Sophia Antipolis, 1996.

[Agh86] Gul A. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

[Ama94] Roberto M. Amadio. Translating core Facile. Technical Report ECRC-TR-3-94, European Computer-Industry Research Center, GmbH, Munich, 1994. Also available as a technical report from CRIN(CNRS)-Inria (Nancy).

[AP94] Roberto M. Amadio and Sanjiva Prasad. Localities and failures. Technical Report ECRC-M2-R10, European Computer-Industry Research Center, GmbH, Munich, 1994.

[Bar92] Henk Barendregt. Lambda calculi with types. In Gabbay Abramsky and Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

[BMT92] Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. In *ACM Principles of Programming Languages*, January 1992.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus (note). Rapporte de Recherche 1702, INRIA Sofia-Antipolis, May 1992.

[Bru94] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2), April 1994. A preliminary version appeared in POPL 1993 under the title "Safe Type Checking in a Statically Typed Object-Oriented Programming Language".

[Car84] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984. Full version in *Information and Computation* 76(2/3):138–164, 1988.

[Car86] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, pages 21–47. Springer-Verlag, 1986. Lecture Notes in Computer Science No. 242.

[Car90] Luca Cardelli. Notes about $\mathrm{F}^{\omega}_{<:}$. Unpublished manuscript, October 1990.

[Car91] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*. Springer-Verlag, 1991. An earlier version appeared as DEC Systems Research Center Research Report #45, February 1989.

[Car95] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995. Short version in *Principles of Programming Languages (POPL)*, January 1995.

[CC91] Felice Cardone and Mario Coppo. Type inference with recursive types. Syntax and semantics. *Information and Computation*, 92(1):48–80, 1991.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, CA, January 1990. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[CMMS94] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994. A preliminary version appeared in TACS '91 (Sendai, Japan, pp. 750–770).

[Com94] Adriana B. Compagnoni. Decidability of higher-order subtyping with intersection types. In *Computer Science Logic*, September 1994. Kazimierz, Poland. Springer *Lecture Notes in Computer Science* 933, June 1995. Also available as University of Edinburgh, LFCS technical report ECS-LFCS-94-281, titled "Subtyping in $F^{\omega}_{\wedge}$ is decidable".

[CS87] G. Costa and C. Stirling. Weak and strong fairness in CCS. *Information and Computation*, 73(3):207–244, 1987.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Principles of Programming Languages*, January 1996.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421, Pisa, Italy, August 1996. Springer-Verlag. LNCS 1119.

[FM94] Kathleen Fisher and John Mitchell. Notes on typed object-oriented programming. In *Proceedings of Theoretical Aspects of Computer Software, Sendai, Japan*, pages 844–885. Springer-Verlag, April 1994. LNCS 789.

[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.

[GMP89] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A Symmetric Integration of Concurrent and Functional Programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the Twenty-First ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 123–137, Portland, OR, January 1994.

[Hol83] Sören Holmström. PFL: A functional language for parallel programming, and its implementation. Programming Methodology Group, Report 7, University of Goteborg and Chalmers University of Technology, September 1983.

[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.

[HP95] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995. Previous versions appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, (pages 251–262) and, under the title "An Abstract View of Objects and Subtyping (Preliminary Report)," as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 512 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1991.

[HY94] Kohei Honda and Nobuko Yoshida. Combinatory representation of mobile processes. In Principles of Programming Languages (POPL), pages 348–360, January 1994.

[JGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 295–308, St. Petersburg, Florida, January 21–24, 1996. ACM Press.

[Jon93] Cliff B. Jones. A pi-calculus semantics for an object-based design notation. In E. Best, editor, *Proceedings of CONCUR'93*, LNCS 715, pages 158–172. Springer-Verlag, 1993.

[Kob96] Naoki Kobayashi. *Concurrent Linear Logic Programming.* PhD thesis, Department of Information Science, University of Tokyo, April 1996.

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

[KY94] Naoki Kobayashi and Akinori Yonezawa. Type-theoretic foundations for concurrent object-oriented programming. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)*, pages 31–45, 1994.

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 142–153, San Francisco, California, January 1995.

[Mat91] David Matthews. A distributed concurrent implementation of Standard ML. Technical Report ECS-LFCS-91-174, University of Edinburgh, August 1991.

[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.

[Mit90] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 109–124, January 1990. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

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

[MPS86] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

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

[Nie95] Oscar Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995. Earlier version in proceedings of *OOPSLA '93*, published in *ACM Sigplan Notices*, 28(10), October 1993, pp. 1–15.

[NP96] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. In *Proceedings of CONCUR '96*, August 1996.

[NSL96] Oscar Nierstrasz, Jean-Guy Schneider, and Markus Lumpe. Formalizing composable software systems — a research agenda. In *Formal Methods in Open, Object-Based Distributed Systems (FMOODS '96)*, February 1996.

[Pie97] Benjamin C. Pierce. Programming in the pi-calculus: A tutorial introduction to Pict. Available electronically, 1997.

[PRT93] Benjamin C. Pierce, Didier Rémy, and David N. Turner. A typed higher-order programming language based on the pi-calculus. In *Workshop on Type Theory and its Application to Computer Systems, Kyoto University*, July 1993.

[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.

[PS97] Benjamin Pierce and Martin Steffen. Higher-order subtyping. *Theoretical Computer Science*, 1997. To appear. A preliminary version appeared in IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET), June 1994, and as University of Edinburgh technical report ECS-LFCS-94-280 and Universität Erlangen-Nürnberg Interner Bericht IMMD7-01/94, January 1994.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[PT95] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan (Nov. 1994)*, number 907 in Lecture Notes in Computer Science, pages 187–215. Springer-Verlag, April 1995.

[PT97] Benjamin C. Pierce and David N. Turner. Pict language definition. Draft report; available electronically as part of the Pict distribution, 1997.

[Rep91] John Reppy. CML: A higher-order concurrent language. In *Programming Language Design and Implementation*, pages 293–259. SIGPLAN, ACM, June 1991.

[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.

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

[San93] Davide Sangiorgi. An investigation into functions as processes. In *Proc. Ninth International Conference on the Mathematical Foundations of Programming Semantics (MFPS'93)*, volume 802 of *Lecture Notes in Computer Science*, pages 143–159. Springer Verlag, 1993.

[San94] Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111(1):120–153, 1994.

[Sew96] Peter Sewell. Observations on Pict, a nondeterministic programming language. Manuscript, 1996.

[Smo94] Gert Smolka. A Foundation for Concurrent Constraint Programming. In *Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, Munich, Germany, September 1994. Invited Talk.

[Smo95] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[TAL90] David Tarditi, Anurag Acharya, and Peter Lee. No assembly required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, November 1990.

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

[Vas94] Vasco T. Vasconcelos. Typed concurrent objects. In *Proceedings of the Eighth European Conference on Object-Oriented Programming (ECOOP)*, volume 821 of *Lecture Notes in Computer Science*, pages 100–117. Springer-Verlag, July 1994.

[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.

[Wal95] David Walker. Objects in the π-calculus. *Information and Computation*, 116:253–271, 1995.