Concurrent Objects in a Process Calculus

Benjamin C. Pierce
Department of Computer Science
University of Edinburgh
`bcp@dcs.ed.ac.uk`

David N. Turner
Department of Computer Science
University of Glasgow
`dnt@dcs.gla.ac.uk`

Abstract

A programming style based on concurrent objects arises almost inevitably in languages where processes communicate by exchanging data on channels. Using the PICT language as an experimental testbed, we introduce a simple object-based programming style and compare three techniques for controlling concurrency between methods in this setting: explicit locking, a standard *choice* operator, and a more refined *replicated choice* operator.

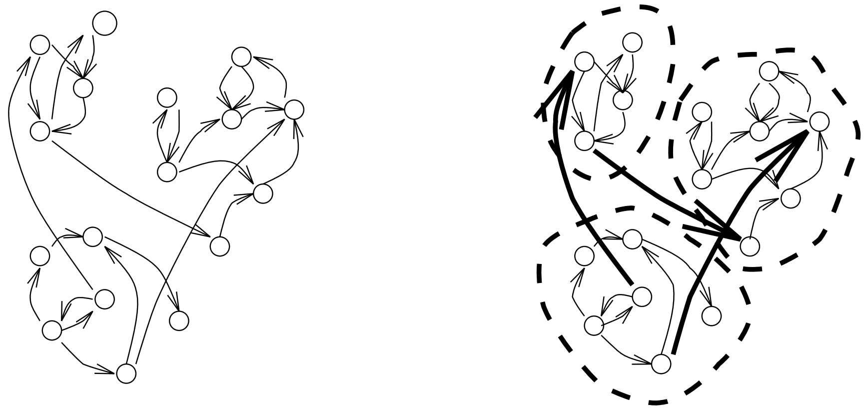
1 Introduction

Process calculi have become a popular vehicle for research on the foundations of concurrent object-oriented programming languages [Pap92, Nie92, VT93, Jon93, Wal94, Smo94, KY94, Vas94, etc.]. Their usual role is as a notation for the rigorous explanation of existing high-level language features. We propose that process calculi may also be viewed as design tools for high-level structures. approach. Beginning from the observation that objects arise naturally from the requirements of programming in a standard process calculus, we consider the suitability of various idioms for defining the behavior of objects in this setting. For this investigation we use a fragment of PICT, a programming language based on Milner, Parrow, and Walker's π-calculus [MPW92, Mil91].

1.1 Processes as Objects

The π-calculus is the best-known member of a family of "mobile" process calculi [MT91, Bou92, HT91, etc.], in which communication topologies can evolve dynamically during evaluation. The primitive constructs in these systems include sending data along a channel, receiving data from a channel, creating channels, and running processes in parallel. Communication along channels is the primary means by which computations progress. Channels are created dynamically and may be passed as data along other channels. Nondeterminism arises from multiple senders or receivers trying to use the same channel simultaneously.

In such calculi, a concurrent system may be thought of as a kind of "process community" (or "process soup"): an unstructured collection of autonomous agents communicating in arbitrary patterns over channels (c.f. left-hand side of Figure 1.1). In practice, however, programs contain significantly more structure

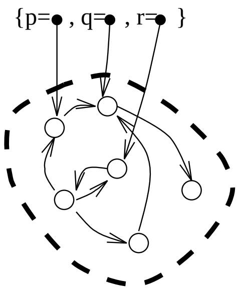


than this picture would suggest. In particular, their correct behavior usually depends on the maintainence of numerous *invariants*. To control the complexity of the task of programming, these invariants tend to be maintained *locally* by small groups of processes that cooperate in presenting to the rest of the world some abstraction of a shared internal state. Given the existence of such groupings, it is tempting to speak of them as distinct units, and of communications across group boundaries as messages from one *group* to another (right-hand side of Figure 1.1).

1.2 Objects as Records

In a pure process calculus, these groups of processes exist nowhere but in the programmer's mind. But adding only a little more structure — the ability for a program to refer explicitly to groups of processes — suffices to give them the status of rudimentary *objects*.

In the π-calculus, there is no way for one process to directly affect or even refer to another process; it can only send messages on some channel where, by convention, the other process listens. By analogy, referring to a group of processes consists exactly in referring to a collection of channels where these processes are listening. The standard notion of *records* is attractive for this purpose, since it allows members of a group of channels to be selected by name.

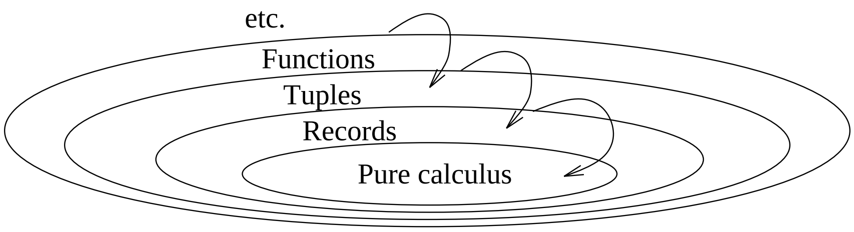


Packaging a set of "service ports" together in a record provides a simple, well-defined interface through which a set of related services may be accessed. Moreover, this packaging promotes a *higher-order* style of programming with objects, since the complete interface to an object may be manipulated as a single value.

Some variants of the π-calculus have included basic facilities for manipulating collections of channels. For example, Milner's *polyadic* π-calculus [Mil91] includes simple syntax for constructing and transmitting tuples of channels. Operations on tuples may easily be encoded using the lower-level ("monadic") communication facilities, so the polyadic π-calculus can be regarded as a simple syntactic sugaring of a monadic calculus. Similarly, we may add special syntax for records of channels without fundamentally changing the nature of the calculus.

1.3 Derived Forms and Typing

We can go on to provide many more layers of derived forms to ease the programmer's task. For example, the π-calculus can be used to encode algebraic datatypes [Mil91], higher-order processes [San92], and even λ-expressions [Mil90].



If we only consider the abstract behaviour of programs there is no difference in expressive power between these layers. For simplicity, we might therefore choose the smallest calculus as our theoretical core. However, we are interested in strongly typed programming languages, and this brings other concerns into play. Ideally, the types of higher-level forms should be derived from the translations: to typecheck a high-level program, one should simply translate it and then typecheck according to the rules of the lower-level system. From this perspective, the smallest π-calculus is too homogeneous: the translations of records and tuples into the pure ("monadic") π-calculus do not preserve enough structure to give rise to the expected typing rules.

Another concern that may influence our choice of core calculus is efficiency. In the design of the PICT core language, we have retained both tuples and records, even though tuples and their typing rules *can* be satisfactorily encoded in a core calculus with just records. Tuples are used very frequently in our programs and cannot be implemented in terms of records with acceptable efficiency using current compilation technology, since we allow single-field extension of records and hence the relative positions of fields are not necessarily known at compile time.

1.4 Synchrony vs. Asynchrony

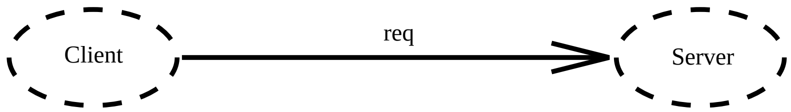
The fundamental communication primitive in most process calculi is *synchronous rendez-vous*, in which both sender and receiver are blocked until the communication occurs. For example, in the process expression

```
(c!v > d!w > P)  |  (c?x > d?y > Q)
```

the output expression `d!w > P` cannot proceed until the preceding output on `c` has completed; similarly, the input expression `d?y > Q` cannot proceed until the value `x` has been received from `c`. Thus, the whole parallel composition has only one possible communciation sequence:

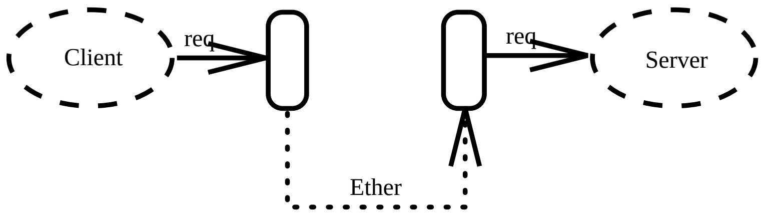
$$\begin{array}{ll} & \texttt{(c!v > d!w > P)} \;\;|\;\; \texttt{(c?x > d?y > Q)} \\ \longrightarrow & \texttt{d!w > P} \;\;|\;\; \{\texttt{x} \mapsto \texttt{v}\}\texttt{Q} \\ \longrightarrow & \texttt{P} \;\;|\;\; \{\texttt{y} \mapsto \texttt{w}\}\{\texttt{x} \mapsto \texttt{v}\}\texttt{Q} \end{array}$$

In principle, the fact that output is synchronous enables a client object to tell when a request has been received by some other object that is acting as a server.



For example, if the request involves some update of the server's state, then the fact that it has been accepted may be used by the client as a signal that it is safe to make further requests of the same server, if it knows that the server will actually perform the update before accepting any more communications.

However, such reasoning is prone to error, since it is invalidated by the simple act of interposing a buffer between client and server. For example, if the client and server are sometimes run on separate machines, we may need to add a *surrogate* object on the client's machine, which forwards requests to the server's machine.



Now any assumptions that the client makes about the state of the server may be incorrect, since a synchronization on the client-side "req" channel indicates only that the surrogate, not the server, has received the request.

To guard against such errors, we allow only asynchronous output in PICT, essentially restricting the "continuation" of each output expression to the null process. This forces the programmer to use an explicit acknowledgement signal from server to client, eliminating any possible sensitivity to buffering of requests.

Even in a language with synchronous communication, explicit acknowledgements are necessary in most client-server interactions, either because some time may elapse between the receipt of a request and the completion of its processing, or because some result value must be sent back to the client. In the common case where the client must wait for the server's response before proceeding, the synchrony or asynchrony of communication of the request is irrelevant.

1.5 The Pict Language

The foregoing considerations have strongly influenced the design of PICT, a high-level concurrent language in the tradition of ML. Like ML, PICT is constructed from a tiny core calculus — an asynchronous, choice-free π-calculus with tuples and extensible records (we discuss the omission of choice in Section 6.2) — plus a variety of higher-level features defined by simple encodings into the core. Also like ML, the core calculus, and hence the derived forms, are statically typed. A uniprocessor implementation is publically available.

The full PICT language and type system are described elsewhere [Pie94, PT94a]. In this paper, we introduce just those constructs needed for the development of the examples; in particular, we shall not discuss the type system further. Most of the examples shown here do actually typecheck in the current version of PICT; in a few places, however, typing annotations must be added to help the partial type inference algorithm.

1.6 Outline of the Remainder

Section 2 defines the core language of PICT and its operational semantics. Section 3 shows how rudimentary objects may be implemented directly in this low-level calculus, using explicit locking to protect internal invariants. Section 4 introduces some higher-level derived forms to improve readability; these are used in Section 5 to develop additional examples of objects based on explicit locking. Section 6 introduces a higher-level programming idiom in which objects may have many internal states, each accepting a different set of messages; the basis of this construction is a library module implementating the familiar external *choice* operator. Section 7 proposes a more appropriate abstraction, called *replicated choice*, and shows how to use it to express earlier examples. Section 8 considers some abstraction mechanisms that allow synchronization code to be reused.

2 Core Language

We now proceed to a rigorous definition of the syntax and semantics of the core of PICT.

2.1 Syntax

The entities that can be communicated on channels are called *values*. They include channels (this is what makdes PICT and the π-calculus *mobile* calculi) as well as tuples and records.

```
Val = Id                                         variable
      [Val,...,Val]                              tuple
      record end                                 empty record
      Val with Id=Val end                        record extension
```

As in the π-calculus, there are no channel constants in the concrete syntax, only variables ranging over channels. Record values are constructed by incrementally extending the empty record with single fields. We allow the usual multi-field form `record` l_1=v_1,...,l_n=v_n `end` as an abbreviation for `record end with` l_1=v_1 `end ... with` l_n=v_n `end`.

If `v` is a record that already contains a field with the label `l`, the record extension operator `v with l=v' end` creates a copy of `v` where the field labelled `l` is modified to contain the value `v'`. If `l` is not present in `v`, it creates a copy of `v` extended with a new field. For example,

```
record x=3, y=4 end with z=5 end
    = record x=3, y=4, z=5 end
record x=3, y=4 end with x=0 end
    = record x=0, y=4 end
```

Values are decomposed by means of *patterns* in input prefixes.

```
Pat = Id                                         variable pattern
      [Pat,...,Pat]                              tuple pattern
      record Id=Pat,...,Id=Pat end               record pattern
      _                                          wildcard pattern
```

For example, the process expression

```
x?[y,z] > e
```

reads a pair from the channel `x`, matches it against the pattern `[y,z]`, and continues with `e`, where `y` is bound to the left-hand side of the pair and `z` is bound to the right-hand side. Placing this expression in parallel with a corresponding output yields the reduction

$$\begin{array}{ll} & \texttt{(x?[y,z] > e) | x![a,[]]} \\ \longrightarrow & \{\texttt{y} \mapsto \texttt{a}\}\{\texttt{z} \mapsto \texttt{[]}\}\texttt{e} \end{array}$$

Wildcard patterns are convenient when we wish to ignore some part of a value that we are matching against. For example, matching the value `[3,[]]` against the pattern `[_,y]` results in `y` being bound to `[]`.

A process prefixed by a pattern is called an *abstraction*.

```
Abs = Pat > Proc                            abstraction
```

Introducing abstractions as a separate syntactic class, even though they are quite simple, gives some "growing room" to the syntax. We make use of this in the full language to allow higher-order functions to appear wherever process abstractions are allowed.

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

```
Proc = Val?Abs                              input prefix
       Val?*Abs                             replicated input
       Val!Val                              output atom
       Proc|Proc                            parallel composition
       let Dec in Proc end                  declaration
```

The replicated input prefix is a combination of ordinary input with the *replication* operator (usually written `!Proc`) of the π-calculus. We restrict replication to this form for reasons of efficiency; the unrestricted replication operator can be encoded using only replicated input. In the concrete syntax, the scope of an abstraction extends as far to the right as possible, so that, for example, `c?x > x![] | a!b` is parsed as `c?x > (x![] | a!b)`.

A `new` declaration introduces a fresh channel:

```
Dec = new Id                                channel creation
```

Again, `Dec` is provided as a separate syntactic category so that other kinds of declarations can be added as derived forms. For convenence the compiler translates

$$\texttt{let new } \texttt{x}_1\texttt{,...,x}_n \texttt{ in e end}$$

into:

$$\texttt{let new x}_1 \texttt{ in ... let new x}_n \texttt{ in e end ... end}$$

2.2 Semantics

The operation semantics of PICT programs is presented in two steps. First, we define a *structural congruence* relation $\mathtt{e}_1 \equiv \mathtt{e}_2$ on process expressions; 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* $\mathtt{e}_1 \rightarrow \mathtt{e}_2$ on process expressions, specifying how processes evolve by means of communication.

Structural congruence plays an important technical role as a device for simplifying the statement of the reduction relation. For example, the processes `x!v | (x?y > e)` and `(x?y > e) | x!v` both reduce to `[v/x]e`. Since these two are structurally congruent, it suffices to write the reduction rule just for the first case and stipulate, in general, that if a process expression `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, often called the rule of *scope extrusion* in the π-calculus literature, plays a crucial role in the treatment of channels:

$$\frac{\mathtt{x} \notin FV(\mathtt{e}_2)}{\mathtt{let\ new\ x\ in\ e}_1\ \mathtt{end} \mid \mathtt{e}_2 \equiv \mathtt{let\ new\ x\ in\ e}_1 \mid \mathtt{e}_2\ \mathtt{end}} \ \text{(STR-EXTRUDE)}$$

Informally, this rule says that a declaration can always be moved toward the root of the abstract syntax tree ("always," because the preconditions are always satisfied when the rule is read from left to right). For example, the process expression

```
let new y in x!y end | (x?z > z![])
```

may be transformed to:

```
let new y in x!y | (x?z > z![]) end
```

It is precisely this rule that allows the new channel `y` to be communicated outside of its original scope.

A *substitution* is a finite map from variables to values. The empty substitution is written $\{\}$. A substitution mapping the variable `x` to the value `v` is written $\{\mathtt{x} \mapsto \mathtt{v}\}$. 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 σ can be extended to a function from values to values by applying σ to variables that fall in its domain and leaving the rest of the value unchanged. For example, applying the substitution $\sigma = \{\mathtt{x} \mapsto \mathtt{a}\} \cup \{\mathtt{y} \mapsto \mathtt{[]}\}$ to the value `[b,[x],x,y]`, written σ(`[b,[x],x,y]`), yields the value `[b,[a],a,[]]`.

When a value `v` is successfully matched by a pattern `p`, the result is a substitution *match*(`p`, `v`), defined formally as follows:

$$
\begin{array}{lcl}
\mathit{match}(\mathtt{x},\ \mathtt{v}) & = & \{\mathtt{x} \mapsto \mathtt{v}\} \\
\mathit{match}(\mathtt{[p_1,\ldots,p_n]},\ \mathtt{[v_1,\ldots,v_n]}) & = & \mathit{match}(\mathtt{p_1},\ \mathtt{v_1}) \cup \cdots \cup \mathit{match}(\mathtt{p_n},\ \mathtt{v_n}) \\
\mathit{match}(\_,\ \mathtt{v}) & = & \{\,\} \\
\mathit{match}(\mathtt{record\ end},\ \mathtt{record\ end}) & = & \{\,\} \\
\multicolumn{3}{l}{\mathit{match}(\mathtt{record\ j_1{=}p_1,\ldots,j_n{=}p_n,l{=}p\ end},\ \mathtt{v_0\ with\ l{=}v\ end})} \\
\multicolumn{3}{l}{\quad = \quad \mathit{match}(\mathtt{p},\ \mathtt{v}) \cup \mathit{match}(\mathtt{record\ j_1{=}p_1,\ldots,j_n{=}p_n\ end},\ \mathtt{v_0})} \\
\multicolumn{3}{l}{\mathit{match}(\mathtt{record\ j_1{=}p_1,\ldots,j_n{=}p_n\ end},\ \mathtt{v_0\ with\ l{=}v\ end})} \\
\multicolumn{3}{l}{\quad = \quad \mathit{match}(\mathtt{record\ j_1{=}p_1,\ldots,j_n{=}p_n\ end},\ \mathtt{v_0}), \quad \text{if } \mathtt{l} \notin \{\mathtt{j_1,\ldots,j_n}\}}
\end{array}
$$

The last three clauses specify how record values are matched against record patterns: if the final field in the record appears in the pattern, it is added to the results of the match; if not, it is discarded. Either way, the matching process continues with the rest of the fields until the pattern becomes empty, at which point it succeeds. If `v` and `p` do not have the same structure (or if both are records but `v` is missing a field), then *match*(`p`, `v`) is undefined.

The reduction relation $\mathtt{e} \rightarrow \mathtt{e}'$ may be read as "The process `e` can evolve to the process $\mathtt{e}'$." 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:

$$\frac{\mathit{match}(\mathtt{p},\ \mathtt{v}) \text{ defined}}{\mathtt{x!v\ \mid\ (x?p\ >\ e)} \rightarrow \mathit{match}(\mathtt{p},\ \mathtt{v})(\mathtt{e})} \qquad (\textsc{Red-Comm})$$

Similarly, when a replicated input prefix meets an output atom, the result is a running instance of the input's body *plus* a fresh copy of the replicated input itself:

$$\frac{\mathit{match}(\mathtt{p},\ \mathtt{v}) \text{ defined}}{\mathtt{x!v\ \mid\ (x?{*}p\ >\ e)} \rightarrow \mathit{match}(\mathtt{p},\ \mathtt{v})(\mathtt{e})\ \mathtt{\mid\ (x?{*}p\ >\ e)}} \qquad (\textsc{Red-RComm})$$

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

$$\frac{\mathtt{e} \rightarrow \mathtt{e'}}{\mathtt{let\ d\ in\ e\ end} \rightarrow \mathtt{let\ d\ in\ e'\ end}} \qquad (\textsc{Red-Dec})$$

$$\frac{\mathtt{e_1} \rightarrow \mathtt{e_1'}}{\mathtt{e_1\ \mid\ e_2} \rightarrow \mathtt{e_1'\ \mid\ e_2}} \qquad (\textsc{Red-Par})$$

The body of an input expression, on the other hand, *cannot* participate in reductions until after the input has been discharged.

The structural congruence relation captures the distributed nature of reduction. Any two subprocesses at the "top level" of a process expression may be brought into proximity by structural manipulations and allowed to interact.

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

3 Objects with Locks

The following program fragment, implementing integer reference cell objects, illustrates most of the features of the core language:

```
ref?*[init, res] >
  let
    new contents, s, g
  in
    contents!init
  | res!record set=s, get=g end
  | (s?*[v, c] > contents?_ > contents!v | c![])
  | (g?*[r]    > contents?x > contents!x | r!x)
  end
```

This process defines a "function" `ref` — a process that repeatedly listens on the channel `ref` for a pair of parameters: `init`, the initial value to be stored in the cell, and `res`, a "result channel" on which the newly constructed reference cell object is to be returned. When it is invoked by another process sending a pair of values on `ref`, it creates three new channels: `contents`, `s`, and `d`. The channel `contents` acts as a container for the current value of the reference cell. Sending a value on `contents` represents the action of placing a value in the container. Receiving a value from `contents` empties the container; it is then the receiver's responsibility to refill the container by transmitting a new value along `contents`. The container is initialized by sending `init` along `contents`. In parallel with initializing `contents`, `ref` returns on `res` a record containing `s` and `d`. The channels `s` and `d` are the `set` and `get` service ports for the new object. Finally, `ref` starts two server processes, one handling requests on `s` and one handling requests on `d`. Each server waits for a request on its service port, reads `contents` to obtain the current value of the cell, refills `contents` as appropriate, and sends a result or acknowledgement to the client.

Here is a process that interacts with `ref`. It creates a fresh channel `r`, upon which it will receive the new object, and sends to `ref` the initial value, `3`, paired with `r`. In parallel, it waits for `ref`'s reply along `r`:

```
let
  new r
in
  ref![3,r]
| (r?myRef > ...)
end
```

Similarly, to get the current value of a reference cell, we create a fresh reply channel `r`, send `r` along the object's `get` channel, and wait in parallel for the result.

4 Derived Forms

In Section 2, we mentioned that the compiler actually accepts a richer syntax than the pure core language, desugaring higher-level forms (like `new` declarations with multiple identifiers) into expressions of the core. The full PICT language uses several layers of desugarings to provide a high-level syntax reminiscent of the core language of Standard ML [MTH90]. The complete set of derived forms is listed in Appendix A; here, we explain just those used in the examples.

First, PICT allows conditional process expressions like `if v then e else f`, where `v` must evaluate to a boolean value.

More interestingly, a variety of *complex values* are supported, including `let` and `if` expressions whose bodies are values, and record selection `v.l` (which extracts the field labelled `l` from `v`). When a complex value appears in a position where a simple (core language) value is expected, the compiler generates a process that evaluates the complex value and sends the result over a channel. For example, if `v` is a complex value, then `x!v` is transformed to:

```
let
  new t
  run <compute v and send the result on t>
in
  t?r > x!y
end
```

Special syntax for standard primitive types like numbers and strings is also provided.

We allow a number of additional forms of declaration clause besides `new`. The sequence of declarations `let` d_1 d_2 `in e end` is equivalent to the nested declaration `let` d_1 `in let` d_2 `in e end end`. A `run` declaration introduces a process expression that is executed in parallel with the body of the declaration: `let run e in f end` is equivalent to `e | f`.

Recursive definitions of processes are extremely common in PICT. The declaration `def x a` introduces a process definition, where `a` is a process abstraction (a pattern and process expression, separated by a `>` symbol). The abstraction `a` can refer to `x` recursively. The declaration `def d x > e` is translated as follows:

```
new d
run d?*x > e
```

More generally, a set of mutually recursive declarations (where both `e` and `f` may refer to `c` and `d`) can be written

```
def c x > e
and d y > f
```

and is equivalent to:

```
new c,d
run c?*x > e
run d?*y > f
```

Using these forms, the `ref` example can be rewritten as:

```
def ref [init, res] >
  let
    new contents
    run contents!init
    def s [v, c] > contents?_ > contents!v | c![]
    def g [r]    > contents?x > contents!x | r!x
  in
    res!record set=s, get=g end
  end
```

We can even avoid having to make up names for the definitions `s` and `g` by substituting the definitions themselves for the channels `s` and `g` in the record `record set=s, get=g end`. This makes `set` and `get` look more like method definitions.

```
record
  set = abs [v, c] > contents?_ > contents!v | c![] end,
  get = abs [r]    > contents?x > contents!x | r!x end
end
```

A value expression of the form `abs [x,y,z] > e end` is called an *anonymous abstraction*; it is equivalent to `let def t [x,y,z] > e in t end`.

Lambda-calculus-style function definition and application are also supported, using the special form of abstraction `[`p_1`,...,`p_n`] = v` with the left-hand side a tuple of patterns and the right-hand side a value instead of a process; the two are separated by `=` instead of `>`. This is translated into `[`p_1`,...,`p_n`,r] > r!v`, which uses the implicit parameter `r` to return the value on the right-hand side. So, for example, the definition

```
def d[x] = [x,x,x]
```

becomes:

```
def d[x,r] > r![x,x,x]
```

The dual operation of function application is supported by translating value expressions of the form `f[`a_1`,...,`a_n`]` into expressions that explicitly construct a new result channel `r`, pass it to `f` along with the arguments $a_1, \ldots, a_n$, and wait for the result on `r`. The result of the overall expression is whatever is received on `r`.

The declaration `val p = v` matches the value `v` against the pattern `p`. Thus, pattern matching the result of applying `d` to `7` binds `x`, `y` and `z` to `7`:

```
let
  val [x,y,z] = d[7]
in ... end
```

This program is translated as:

```
let
  new r
  run d![7,r]
in
  r?[x,y,z] > ...
end
```

We often execute "functions" which perform some operation and then return the trivial value `[]` to indicate that they have finished. The expression `f[x]; e` is equivalent to `let val [] = f[x] in e end`. It waits for the function application `f[x]` to signal completion and then executes `e`.

Infix operators are provided by a simple extension of the parser: if the name of a channel is symbolic rather than alphanumeric, then it can be applied to a pair of arguments by placing it between them. For example, `a+b` means the same as `(+)[a,b]` (placing `+` in parentheses takes away its infix status).

As a function, the `ref` constructor looks like this:

```
def ref [init] = let
  new current
  run current!init
in
  record
    set = abs [v, c] > current?_ > current!v | c![] end,
    get = abs [r]    > current?x > current!x | r!x end
  end
end
```

To use a reference cell in a program, we create it by applying `ref` to an initial value, `0`, and then send it `set` and `get` messages by projecting the appropriate fields:

```
val r = ref[0]
val v1 = r.get[]
val _ = r.set[5]
val v2 = r.get[]
```

The PICT runtime system provides a special channel `printi` that prints integer values on the standard output stream:

```
run printi!v2
```

Running all these clauses together produces the expected output:

```
5
```

Of course, this version of `ref` is still fairly low-level. We could continue the process of adding higher-level syntactic forms, but we have already reached the point where the behavior of the program is not greatly obscured by low-level details, and this is enough for our purposes. Our goal in this paper is to experiment with different styles of behavior for objects; if we find one that we like, we may later consider "canonizing" it as an official, recommended idiom by providing special syntax.

5 More Objects With Locks

Other kinds of reference cells may be implemented almost as easily as `ref`. For example, here is a "null reference cell" that starts out in an uninitialized state instead of being given a starting value at creation time:

```
def nullRef [] = let
  new contents, set
  run set?[v,c] > contents!v | c![] |
    (set?*[c,v] > contents?_ > contents!v | c![])
in
  record
    set = set,
    get = abs [r] > contents?x > r!x | contents!x end
  end
end
```

The `get` method is implemented exactly as before: it receives a request, reads a value from the internal channel `contents` (thus blocking out any other instances of `set` or `get` that are also trying to read the current contents), returns a result to the process that invoked `get`, and restores `contents` by writing the same value. The `set` method, on the other hand, is prefixed with a new clause that reads a single request from `set`, sends the given value along `contents` *without* first reading a value from `contents`, and then enters a state where it behaves like the `set` of ordinary `ref` cells. Notice that `nullRef` returns an object with the same interface as `ref`; the only difference that can be detected by clients is that all `get` requests to a `nullRef` will block until the first `set` request has been processed.

Another variation on reference cells adds a `quit` method that grabs the current contents and fails to restore it, rendering the object unable to process any further requests.

```
def quitRef[init] = let
  new contents
  run contents![]
in
  record
    set = abs [v,c] > contents?_ > contents!v | c![] end,
    get = abs [r] > contents?x > r!x | contents!x end,
    quit = abs [c] > contents?_ > c![] end
  end
end
```

Finally, here is a more sophisticated refinement of reference cells, supporting an extra pair of messages, `init` and `clear`. Like the instances of `nullRef`, a `clearRef` cell starts life in an uninitialized state. But instead of waiting for a first `set` message, it delays both `set` and `get` messages until it receives an `init`. It then accepts `set` and `get` messages until it receives a `clear`, which sends it back to the initial state, listening only for `init`.

```
def clearRef [] = let
  new contents, clear, init
  def server[] >
    init?[x,c] > c![] | contents!x |
    (clear?[c] > contents?_ > c![] | server![])
  run server![]
in
  record
    set = abs [v,c] > contents?_ > contents!v | c![] end,
    get = abs [r] > contents?x > r!x | contents!x end,
    clear = clear,
    init = init
  end
end
```

The alternation between `init` and `clear` is handled by a single process. The auxiliary definition `server` implements a tail-recursive loop whose body first accepts `init`, then accepts `clear`, and then restarts `server`. The `set` and `get` methods are identical to the ones in our original `ref` cells.

Notice that this implementation of `clearRef` is somewhat unsatisfying: it describes an object with two distinct internal states, but this fact is not apparent from the organization of the code. The `server` loop must be examined carefully in order to decide when `set` and `get` messages will be accepted. We shall return to this point in Section 6.3.

6 Choice

The examples from Sections 3 and 5 illustrate how channels can be used as repositories for shared state, yielding a simple mechanism for ensuring mutual exclusion between method bodies. Another obvious means of selectively processing messages is a *choice* (or *summation*) operator. Indeed, such operators are often assumed as basic in foundational models of concurrency.

In PICT, choice is not part of the core language, but is available as a library module. Instead of writing `(c?x > e) + (d?y > f)`, we write:

```
sync!(
  c => abs x > e end
$
  d => abs y > f end
)
```

This expression may perform an input on either `c` or `d`, but not both: if the first input succeeds, the second clause is thrown away, and vice versa. Informally, `c=>abs x > e end` is like the ordinary input clause `c?x > e`, except that, when it is composed with other such clauses using the choice operator `$`, only the first one to become ready will actually proceed. The `sync` operator takes a list of clauses built in this way (i.e. a description of a choice to be performed) and yields a process (the running choice itself).

6.1 Objects with Choice

Using choice, all the service ports of an object may be handled by a single server process of the form, yielding a style of object definition similar to Actors [Hew77, Agh86]:

```
def server[] >
  sync!(
    c=>abs x > ... <handle request c> ... server![] end
  $
    d=>abs y > ... <handle request d> ... server![] end
  )
```

For example, we can use choice to implement reference cell objects as follows:

```
def ref[init] = let
  new set,get
  def server x >
    sync!(
      get=>abs [r] > r!x | server!x end
    $
      set=>abs [v,c] > c![] | server!v end
    )
  run server!init
in
  record set = set, get = get end
end
```

The current contents of the cell are held in the server parameter `x`. The `set` method restarts the server with the new value `v`, while the `get` method passes on the old contents.

6.2 Implementing Choice

One of the reasons that we avoided building the choice operator into PICT since it is actually possible to implement it as a library module (see Section 6 for more details). This has the useful effect of simplifying the representation of channels in our runtime system. In PICT it is very important that communication is as cheap as possible, since communication is our only means of computation. Many channels, such as those involved in encoding functions as processes, are never used with the choice operator, so it is important that we do not pay the cost of using choice on every channel that we create. This section presents a simple implementation of choice.

The basic abstraction used in the implementation is a *lock*. A lock is queried by sending it a boolean result channel. The first time this happens, it sends back the value `true` over the result channel; for all future requests, it returns the value `false`. In other words, a process that tries to "obtain" a lock by querying its value receives `true`, indicating that it has succeeded; all subsequent tries fail.

```
def newLock [] = let
  new lock
  run lock?[r] > r!true | (lock?*[r] > r!false)
in
  lock
end
```

In essence, there is a race going on between all of the receivers in a choice. The first one that is ready to proceed is allowed to do so; the rest become garbage. This race is mediated by a single lock, which is created by the `sync` operation and distributed to all the individual receiver clauses.

It is important to distinguish the description of a choice from the running choice process that is created when `sync` is applied to this description. Following Reppy [Rep92, Rep88, Rep91], we call the description of a choice an *event*. The `$` operation takes two events and creates a new event that, when it receives its lock, transmits it to both component events.

```
def ($)[e1, e2] =
  abs lock >
    e1!lock | e2!lock
  end
```

The `sync` operation takes an event and triggers it with a freshly created lock.

```
def sync e > e!(newLock[])
```

Finally, the `=>` operation takes a channel `c` and a continuation `receiver` that is waiting to accept a data value over `c`, and returns an event.

```
def (=>) [c, receiver] =
  abs lock >
    c?v >
    if lock[] then
```

```
        receiver!v
      else
        c!v
      end
    end
```

It first waits for a lock; then it waits for a value to arrive on `c`. Once a value `v` arrives, it checks to see whether the lock has already been taken — i.e., whether it has won or lost the race. If it wins (gets the value `true` from the lock), it sends `v` along to the receiver. If it loses, it re-sends `v` along `c`, so that another receiver has the chance to pick it up.

Note, in passing, that the correctness of this simple implementation of events depends crucially on the fact that output in this setting is asynchronous: since the original sender on `c` cannot observe when the value it sent is recieved, we may safely perform a receive, check whether we actually want the value, and resend it if not.

The identifiers `sync`, `=>`, and `$` form part of an events library, which provides selective communication in the style of CML [Rep92, Rep88, Rep91].

6.3 Multi-State Objects

For most of the examples we have considered so far, locks and choice can be used to build implementations of comparable clarity. The explicit locking idiom, being closer to the base language, is more efficient for these examples. However, the more powerful choice operator has its advantages: in particular, it allows us to conveniently express objects whose methods are not always available.

Here is an alternative implementation of the `clearRef` example from Section 5. Like the earlier one, it has two internal states: an `empty` state where it can only receive an `init` message, and a `full` state where `get`, `set`, and `clear` are allowed.

```
def clearRef [] = let
  new set,get,clear,init

  def empty [] >
    sync!(
      init => abs [v,c] > full!v | c![] end
    )
  and full x >
    sync!(
      set=>abs [v,c] > full!v | c![] end
    $
      get=>abs [r] > r!x | full!x end
    $
      clear=>abs [c] > c![] | empty![] end
    )
  run empty![]
in
```

```
    record set = set, get = get, clear = clear, init = init end
  end
```

In effect, the alternation between states in the `server` loop of Section 5 has been expanded to include the receivers of all messages.

6.4 Efficiency

The simple implementation of events described in Section 6.2 has a fairly serious shortcoming: most programs that use these events to build objects will eventually run out of memory! To see why, consider what happens if a program creates a `clearRef` and repeatedly sends it `set` and `get` messages without ever sending a `clear`. Each time the server loop `full` is executed, a fresh lock is created and passed to the three receivers. But the receiver for `create`, like the other two, does not actually read the lock until it has first read a data value from the channel `create`; only at this point does it read pair of a lock and a receiver and checking to see whether it has won the race for the lock. So, each time `full` is executed, a new lock/receiver pair is added to a queue in the runtime system. If no data is ever sent on `clear`, this queue will grow until it exhausts the available virtual memory.

The implementation of events in the PICT distribution uses a more complex algorithm to patch this space leak. Each channel on which events are being performed is associated with a *channel manager* process that periodically scans the receiver queue and throws away any potential receivers whose locks have already been grabbed. This reduces the space overhead for a choice-style object to a constant depending on the number of channels involved in each choice.

However, even with this improved implementation, choice remains somewhat expensive. Even if it were pushed down into the core language for maximum efficiency, it is hard to see how it could be implemented without at least one memory allocation each time a choice is evaluated.

Our conclusion is that the choice operator is not well-suited to the communication patterns in object-style programs. It is wasteful to create a possibly large set of potential communications, allow one to proceed, discard the others, execute a short critical section, and then create from scratch another very similar set of potential communications. It would be better to recognize explictly that the *very same* set of potential communications is invoked again and again, only changing the value of the current state that is passed from one instance to the next.

7 Replicated Choice

It is not hard to implement a more refined variant of the choice primitives by extending the simple implementation from Section 6.2. We call this variant *replicated choice*, since each potential communication that is created may be triggered repeatedly. We replace the receiver constructor `=>` by a replicated receiver constructor `==>` and the choice-instantiation operation `sync` by an initialization operation `start`.

Each replicated event is associated with an *event manager* that coordinates the actions of receivers. It is the job of the `start` operation to create this manager.

```
def start [trigger, e] > let
  new state, ready, count, suspended
  run ready!false | count!0

  def manager [restart,nack,ack] >
    ready?b >
    if b then
      state?s > ready!false | ack!s | restart![]
    else
      count?n > ready!false | nack![] | suspended!restart | count!(n+1)
    end

  def unblock n >
    if n == 0 then
      ready!true | count!0
    else
      suspended?restart > restart![] | unblock!(n-1)
    end

  run trigger?*s >
    ready?b > count?n >
    if b then
      fatalError!"Received a trigger on an already triggered event"
    else
      state!s | unblock!n
    end
in
  e!manager
end
```

When `start` is invoked, it is passed a trigger channel `trigger` and a replicated event `e`. It creates an event manager (the definition `manager`) and sends it to `e`; also, it starts a server process listening to the channel `trigger`. Four internal channels are used to coordinate these processes:

> `ready` carries a boolean value: `true` if the event has been triggered since the last time one of its receivers was allowed to proceed, or `false` if it is waiting for a trigger;
>
> when `ready` is `true`, `state` carries the current-state value that was provided when the event was last triggered (`state` carries no value when `ready` is false);
>
> `suspended` is an internal queue of receivers that have received data values and tried unsucessfully to proceed since the last trigger;

count contains the number of receivers in suspended (so that we know how many times we can read from suspended without blocking).

The suspended queue prevents the inactive states of multi-state objects from consuming too much energy attempting to fire: when a replicated receiver obtains a data value, asks the event manager whether it may proceed, and is told "No," it is suspended until the event is triggered again. At this point, all of the receivers blocked on the suspended queue are released (by the process defined by unblock).

The definition of the replicated receiver constructor is comparatively simple:

```
def (==>) [chan, receiver] =
  abs manager > let
    def loop [] >
      chan?v > manager![
        loop,
        abs [] > chan!v end,
        abs s > receiver![v,s] end
      ]
    in
      loop![]
    end
  end
```

After being told what channel to use to communicate with its event manager, a replicated receiver waits (as before) for a data value to arrive along the channel chan. When one does, it sends off a packet of three values to the manager: 1) its own continuation (the server process loop), so that the manager can suspend it if necessary; 2) an abstraction to be invoked if the event is not currently triggered; and 3) another abstraction to be invoked if the event is current triggered. The "not-triggered" (nack) abstraction simply resends the data value on chan so that another receiver can pick it up. The "triggered" (ack) abstraction receives the current state (the one that was used to trigger the event manager) and sends both this and the data value to the body of the receiver.

The choice constructor is exactly the same as before:

```
def ($) [e1, e2] =
  abs manager > e1!manager | e2!manager end
```

Using these primitives, we can build a clearRef cell that looks nearly the same as before. The main difference is that empty and full are now names of channels rather than definitions.

```
def clearRef [] = let
  new get,set,clear,init,empty,full

  run start![empty,
     init ==> abs [[v,c],[]] > c![] | full!v end
  ]
```

```
  run start![full,
     get ==> abs [[r],x] > r!x | full!x end
  $
     set ==> abs [[v,c],x] > c![] | full!v end
  $
     clear ==> abs [[c],x] > c![] | empty![] end
  ]

  run empty![]
in
  record set = set, get = get, clear = clear, init = init end
end
```

8 Abstraction and Code Reuse

Code reuse is a common theme in discussions of object-based programming idioms. In this section, we briefly discuss some simple techniques for reusing the code of object definitions.

We do not discuss more powerful techniques for *inheritance* of method bodies and synchronization code here. Based on previous experience with foundational models of sequential objects [PT94b], we believe that any reasonable high-level scheme for concurrent objects with inheritance can be expressed in the setting we have described. However, inheritance in the presence of concurrency seems necessarily to involve separating the interfaces of objects into external and internal parts (depending on whether messages are being sent by external clients or "through `self`"), and both typing and behavioral issues quickly become complex. See [Vas94] for a recent proposal along these lines and further citations.

8.1 Simple Delegation

Probably the simplest way of reusing object definitions is writing new object constructors that act as "front ends" for other, perhaps more complex, objects. For example, suppose we have already implemented `clearRef` cells. Then `quitRef` can be implemented very easily by creating a `clearRef` cell, invoking its `init` method, and renaming its `clear` method to `quit`:

```
def quitRef [init] = let
  val cr = clearRef[]
  val _ = cr.init[init]
in
  cr with quit = cr.clear end
end
```

Similarly, we can define `nullRef` by creating a `clearRef`, creating a fresh `set` channel, and returning an object with the fresh `set` and a copy of the `clearRef`'s `get`. A small process listens to the new `set`, accepts a single request, performs an `init` of the `clearRef`, and then forwards all further requests to the `set` channel of the `clearRef` itself.

```
def nullRef [] = let
  new set
  val cr = clearRef[]
  run set?[v,c] > cr.init![v,c] | forward![set,cr.set]
in
  record set = set, get = cr.get end
end
```

The definition of the forwarder is:

```
def forward [inChan, outChan] > inChan?*x > outChan!x
```

8.2 Synchronization Skeletons

In PICT's standard libraries, heavy use is made of another abstraction mechanism called *synchronization skeletons*. For example, here is a skeleton for a simple printer object:

```
def printerSkel [doPrint,doQuit] = let
  new lock
  run lock![]
in
  record
    print = abs [c] > lock?[] > doPrint[]; lock![] end,
    quit = abs [c] > lock?[] > doQuit[]; c![] end
  end
end
```

Given a pair of abstractions, `doPrint` and `doQuit`, `printerSkel` creates an object with `print` and `quit` methods. An internal lock is used to prevent simultaneous execution of two methods. Each time a method is invoked, the lock is grabbed and held while the appropriate abstraction is invoked; when it returns, the lock is released. Note that, although we have chosen a simple lock-style synchronization mechanism for this example, we could equally well have used a multi-state replicated event. In general, synchronization skeletons can be used to set up arbitrarily complicated patterns of synchronization. The essential point is to write this code only once.

This skeleton can be refined by adding a "front end" that supplies null behavior for the `quit` operation:

```
def simplePrinterSkel [doPrint] =
  printerSkel[doPrint,abs [] = [] end]
```

Now an object constructor can easily be built by calling the specialized skeleton:

```
def stringPrinter [string] =
  simplePrinterSkel[abs [c] > print!string | c![] end]
```

This constructor can be used to build

```
val printHello = stringPrinter["Hello"]
```

and use

```
val _ = printHello.print[]
val _ = printHello.quit[]
val _ = printHello.print[]
```

objects in the usual way.

Continuing the example, we might further extend `simplePrinterSkel` with a counter that keeps track of the number of times that the `print` method is invoked.

```
def countPrint [doPrint] = let
  new count
  run count!0
in
  simplePrinterSkel[abs [c] > count?x > doPrint[]; count!(x+1) | c![] end]
end
```

In some cases, we can even create an object from a skeleton and extend it with additional methods. For example, instances of `zeroCount` provide a `zero` method that resets the internal counter.

```
def zeroCount [doPrint] = let
  new count
  run count!0
in
  simplePrinterSkel[abs [c] > count?x > doPrint[]; count!(x+1) | c![] end]
  with
    zero = abs [c] > count?_ > count!0 | c![] end
  end
end
```

The synchronization skeleton method works very well in a limited range of cases. It performs best when used to define a large number of similar object constructors, all creating objects with the same methods and the same synchronization behavior, but with differing responses to some messages.

Acknowledgements

The ideas presented here have been refined by enjoyable conversations with Robert Halstead, Martin Müller, John Reppy, and Aki Yonezawa.

This research was carried out at the Laboratory for Foundations of Computer Science, University of Edinburgh, and the Department of Computer Science, University of Glasgow. Pierce was supported by a fellowship from the British Science and Engineering Research Council and by the ESPRIT Basic Research Actions TYPES and CONFER. Turner was jointly supported by Harlequin Limited and the U.K. Science and Engineering Research Council.

References

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

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

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.
Influential description of a method for implementing distributed control structures: the ACTOR paradigm.

[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*, Geneva CH, 1991. Springer-Verlag , Berlin, Heidelberg, New York, Tokyo.

[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.

[KY94] Naoki Kobayashi and Aki Yonezawa. Towards foundations for concurrent object-oriented programming — Types and language design. Submitted for publication, 1994.

[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, University of Edinburgh, October 1991. *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991.

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

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML*. The MIT Press, Cambridge, Massachusetts, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[Nie92] Oscar Nierstrasz. Towards an object calculus. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, Lecture Notes in Computer Science number 612, pages 1–20. Springer-Verlag, 1992.

[Pap92] Michael Papathomas. A unifying framework for process calculus semantics of concurrent object-oriented languages. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, LNCS 612, pages 53–79. Springer-Verlag, 1992.

[Pie94] Benjamin C. Pierce. Programming in the pi-calculus: An experiment in programming language design. Tutorial notes on the Pict language. Available electronically, 1994.

[PT94a] Benjamin C. Pierce and David N. Turner. PICT user manual. Available electronically, 1994.

[PT94b] 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".

[Rep88] John Reppy. Synchronous operations as first-class values. In *Programming Language Design and Implementation*, pages 250–259. SIGPLAN, ACM, 1988.

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

[Rep92] John Reppy. *Higher-Order Concurrency.* PhD thesis, Cornell University, June 1992. Technical Report TR 92-1285.

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

[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.

[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.

[VT93] Vasco T. Vasconcelos and Mario Tokoro. A Typing System for a Calculus of Objects. In *First International Symposium on Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 460–474, November 1993.

[Wal94] David Walker. Algebraic proofs of properties of objects. In *Proceedings of European Symposium on Programming.* Springer-Verlag, 1994.

A Operational Semantics and Derived Forms

A.1 Core Language

A.1.1 Matching

$$
\begin{array}{lcl}
\mathit{match}(\mathtt{x},\ \mathtt{v}) & = & \{\mathtt{x} \mapsto \mathtt{v}\} \\
\mathit{match}(\mathtt{[p_1,\ldots,p_n]},\ \mathtt{[v_1,\ldots,v_n]}) & = & \mathit{match}(\mathtt{p_1},\ \mathtt{v_1}) \cup \cdots \cup \mathit{match}(\mathtt{p_n},\ \mathtt{v_n}) \\
\mathit{match}(\mathtt{\_},\ \mathtt{v}) & = & \{\} \\
\mathit{match}(\mathtt{record\ end},\ \mathtt{record\ end}) & = & \{\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{match}(\mathtt{record\ j_1{=}p_1,\ldots,j_n{=}p_n,l{=}p\ end},\ \mathtt{v_0\ with\ l{=}v\ end}) \\
\quad = \quad \mathit{match}(\mathtt{p},\ \mathtt{v}) \cup \mathit{match}(\mathtt{record\ j_1{=}p_1,\ldots,j_n{=}p_n\ end},\ \mathtt{v_0}) \\
\mathit{match}(\mathtt{record\ j_1{=}p_1,\ldots,j_n{=}p_n\ end},\ \mathtt{v_0\ with\ l{=}v\ end}) \\
\quad = \quad \mathit{match}(\mathtt{record\ j_1{=}p_1,\ldots,j_n{=}p_n\ end},\ \mathtt{v_0}), \quad \text{if } \mathtt{l} \notin \{\mathtt{j_1},\ldots,\mathtt{j_n}\}
\end{array}
$$

A.1.2 Structural Congruence

$$\texttt{e}_1 \ \texttt{|} \ \texttt{e}_2 \equiv \texttt{e}_2 \ \texttt{|} \ \texttt{e}_1 \qquad \text{(Str-Comm)}$$

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

$$\frac{\texttt{x} \notin FV(\texttt{e}_2)}{\texttt{let new x in } \texttt{e}_1 \texttt{ end | } \texttt{e}_2 \equiv \texttt{let new x in } \texttt{e}_1 \texttt{ | } \texttt{e}_2 \texttt{ end}} \qquad \text{(Str-Extrude)}$$

A.1.3 Reduction

$$\frac{\texttt{e}_1 \equiv \texttt{e}_1' \rightarrow \texttt{e}_2' \equiv \texttt{e}_2}{\texttt{e}_1 \rightarrow \texttt{e}_2} \qquad \text{(Red-Str)}$$

$$\frac{\texttt{e}_1 \rightarrow \texttt{e}_1'}{\texttt{e}_1 \ \texttt{|} \ \texttt{e}_2 \rightarrow \texttt{e}_1' \ \texttt{|} \ \texttt{e}_2} \qquad \text{(Red-Par)}$$

$$\frac{\texttt{e} \rightarrow \texttt{e'}}{\texttt{let d in e end} \rightarrow \texttt{let d in e' end}} \qquad \text{(Red-Dec)}$$

$$\frac{match(\texttt{p}, \texttt{v}) \text{ defined}}{\texttt{x!v | (x?p > e)} \rightarrow match(\texttt{p}, \texttt{v})(\texttt{e})} \qquad \text{(Red-Comm)}$$

$$\frac{match(\texttt{p}, \texttt{v}) \text{ defined}}{\texttt{x!v | (x?*p > e)} \rightarrow match(\texttt{p}, \texttt{v})(\texttt{e}) \texttt{ | (x?*p > e)}} \qquad \text{(Red-RComm)}$$

A.2 Derived Forms

A.2.1 Definition: A PICT abstract syntax tree that is generated by the core grammar is called *simple*. We write, for example, $simple(\texttt{v})$ iff $\texttt{v} \in \texttt{Val}$ according to the core grammar. By convention, the metavariable `sv` ranges only over simple values.

A.2.1 Complex Outputs

$$\frac{\text{not } simple(\texttt{v}_i)}{\begin{array}{c}\texttt{sv![sv}_1\texttt{,...,sv}_{i-1}\texttt{,v}_i\texttt{,...,v}_n\texttt{]} \Rightarrow \\ \texttt{let new r in r!v}_i \texttt{ | r?x}_i \texttt{ > sv![sv}_1\texttt{,...,sv}_{i-1}\texttt{,x}_i\texttt{,...,v}_n\texttt{] end}\end{array}} \qquad \text{(Tr-Tuple)}$$

$$\frac{\text{not } simple(\texttt{v}_2)}{\begin{array}{c}\texttt{sv}_1\texttt{!(v}_2\texttt{ with l=v}_3\texttt{ end)} \Rightarrow \\ \texttt{let new r in r!v}_2\texttt{ | r?x > sv}_1\texttt{!(x with l=v}_3\texttt{ end) end}\end{array}} \qquad \text{(Tr-RcdL)}$$

$$\frac{\text{not } simple(\texttt{v}_3)}{\begin{array}{c}\texttt{sv}_1\texttt{!(sv}_2\texttt{ with l=v}_3\texttt{ end)} \Rightarrow \\ \texttt{let new r in r!v}_3\texttt{ | r?x > sv}_1\texttt{!(sv}_2\texttt{ with l=x end) end}\end{array}} \qquad \text{(Tr-RcdR)}$$

$$\mathtt{sv!(let\ d\ in\ v\ end)} \Rightarrow \mathtt{let\ d\ in\ sv!v\ end} \qquad (\textsc{Tr-LetV})$$

$$\frac{\text{not } \mathit{simple}(\mathtt{v})}{\mathtt{sv!(v[v_1,\ldots,v_n])} \Rightarrow \mathtt{let\ new\ r\ in\ r!v\ |\ r?x > x![v_1,\ldots,v_n,sv]\ end}} \qquad (\textsc{Tr-App})$$

$$\begin{gathered}\mathtt{sv!(if\ b\ then\ v_1\ else\ v_2\ end)} \Rightarrow \\ \mathtt{let\ new\ t\ in\ let\ new\ f\ in} \\ \mathtt{primif![b,t,f]\ |\ (t?[] > sv!v_1)\ |\ (f?[] > sv!v_2)\ end\ end}\end{gathered} \qquad (\textsc{Tr-IfV})$$

A.2.2 Declarations

$$\mathtt{let\ run\ e_1\ in\ e_2\ end} \Rightarrow \mathtt{e_1\ |\ e_2} \qquad (\textsc{Tr-Run})$$

$$\mathtt{local\ d_1\ in\ d_2\ end} \Rightarrow \mathtt{d_1\ d_2} \qquad (\textsc{Tr-Local})$$

$$\mathtt{let\ in\ e\ end} \Rightarrow \mathtt{e} \qquad (\textsc{Tr-NullDec})$$

$$\mathtt{let\ d\ end} \Rightarrow \mathtt{let\ d\ in\ skip\ end} \qquad (\textsc{Tr-NullIn})$$

$$\mathtt{new\ x_1,\ldots,x_n} \Rightarrow \mathtt{new\ x_1\ \ldots\ new\ x_n} \quad (n \neq 1) \qquad (\textsc{Tr-News})$$

$$\begin{gathered}\mathtt{let\ def\ x_1\ p_1 > e_1\ and\ \ldots\ and\ x_n\ p_n > e_n\ in\ e\ end} \\ \Rightarrow \\ \mathtt{let\ new\ x_1,\ldots,x_n\ in\ (x_1\ ?*\ p_1 > e_1)\ |\ \ldots\ |\ (x_n\ ?*\ p_n > e_n)\ |\ e\ end}\end{gathered} \qquad (\textsc{Tr-Def})$$

$$\mathtt{let\ d_1\ d_2\ in\ e\ end} \Rightarrow \mathtt{let\ d_1\ in\ let\ d_2\ in\ e\ end\ end} \qquad (\textsc{Tr-DecSeq})$$

$$\mathtt{let\ val\ p = v\ in\ e\ end} \Rightarrow \mathtt{let\ new\ r\ in\ r!v\ |\ r?p > e\ end} \qquad (\textsc{Tr-Val})$$

A.2.3 Abstractions

$$\mathtt{[p_1,\ldots,p_n] = v} \Rightarrow \mathtt{[p_1,\ldots,p_n,r] > r!v} \qquad (\textsc{Tr-VAbs})$$

A.2.4 Values

$$\mathtt{v_1\ InfixId\ v_2} \Rightarrow \mathtt{(InfixId)[v_1,v_2]} \qquad (\textsc{Tr-Inf})$$

$$\mathtt{v_1\ `v`\ v_2} \Rightarrow \mathtt{v[v_1,v_2]} \qquad (\textsc{Tr-InfQ})$$

$$\mathtt{abs\ a\ end} \Rightarrow \mathtt{let\ def\ x\ a\ in\ x\ end} \qquad (\textsc{Tr-AnonAbs})$$

$$\mathtt{v_1;v_2} \Rightarrow \mathtt{let\ val\ [] = v_1\ in\ v_2\ end} \qquad (\textsc{Tr-SemiV})$$

$$\frac{n \geq 1}{\begin{gathered}\mathtt{record\ l_1=v_1,\ \ldots,\ l_n=v_n\ end} \Rightarrow \\ \mathtt{record\ end\ with\ l_1=v_1,\ \ldots,\ l_n=v_n\ end}\end{gathered}} \qquad (\textsc{Tr-NonemptyRcd})$$

$$\frac{n \neq 1}{\begin{array}{c}\texttt{v with l}_1\texttt{=v}_1\texttt{, ..., l}_n\texttt{=v}_n\texttt{ end} \Rightarrow \\ \texttt{v with l}_1\texttt{=v}_1\texttt{ end with ... end with l}_n\texttt{=v}_n\texttt{ end}\end{array}} \qquad (\textsc{Tr-BigRcd})$$

$$\texttt{v with l a end} \Rightarrow \texttt{v with l = let def l a in l end end} \qquad (\textsc{Tr-AbsField})$$

A.2.5 Processes

$$\frac{\text{not } \mathit{simple}(\texttt{v})}{\texttt{v?p > e} \Rightarrow \texttt{let val x = v in x?p > e end}} \qquad (\textsc{Tr-In})$$

$$\frac{\text{not } \mathit{simple}(\texttt{v}_1)}{\texttt{v}_1\texttt{!v}_2 \Rightarrow \texttt{let val x = v}_1\texttt{ in x!v}_2\texttt{ end}} \qquad (\textsc{Tr-Out})$$

$$\texttt{skip} \Rightarrow \texttt{let new x in x![] end} \qquad (\textsc{Tr-Skip})$$

$$\texttt{v;e} \Rightarrow \texttt{let val [] = v in e end} \qquad (\textsc{Tr-Semi})$$

$$\texttt{v;} \Rightarrow \texttt{let val [] = v in skip end} \qquad (\textsc{Tr-SemiSkip})$$

$$\begin{array}{c}\texttt{if b then e}_1\texttt{ else e}_2\texttt{ end} \Rightarrow \\ \texttt{let new t in let new f in} \\ \texttt{primif![b,t,f] | (t?[] > e}_1\texttt{) | (f?[] > e}_2\texttt{) end end}\end{array} \qquad (\textsc{Tr-If})$$

A.2.6 Conditionals

$$\frac{n \geq 2}{\begin{array}{c}\texttt{if b}_1\texttt{ then e}_1\texttt{ elif ... elif b}_n\texttt{ then e}_n\texttt{ else e end} \Rightarrow \\ \texttt{if b}_1\texttt{ then e}_1\texttt{ else if ... else if b}_n\texttt{ then e}_n\texttt{ else e end ... end end}\end{array}} \qquad (\textsc{Tr-Elif})$$

$$\frac{n \geq 2}{\begin{array}{c}\texttt{if b}_1\texttt{ then v}_1\texttt{ elif ... elif b}_n\texttt{ then v}_n\texttt{ else v end} \Rightarrow \\ \texttt{if b}_1\texttt{ then v}_1\texttt{ else if ... else if b}_n\texttt{ then v}_n\texttt{ else v end ... end end}\end{array}} \qquad (\textsc{Tr-ElifV})$$

$$\begin{array}{c}\texttt{if b}_1\texttt{ then e}_1\texttt{ elif ... elif b}_n\texttt{ then e}_n\texttt{ end} \Rightarrow \\ \texttt{if b}_1\texttt{ then e}_1\texttt{ elif ... elif b}_n\texttt{ then e}_n\texttt{ else skip end}\end{array} \qquad (\textsc{Tr-NullElse})$$

$$\begin{array}{c}\texttt{if b}_1\texttt{ then v}_1\texttt{ elif ... elif b}_n\texttt{ then v}_n\texttt{ end} \Rightarrow \\ \texttt{if b}_1\texttt{ then v}_1\texttt{ elif ... elif b}_n\texttt{ then v}_n\texttt{ else [] end}\end{array} \qquad (\textsc{Tr-NullElseV})$$