A Concurrency Monad Based on Constructor Primitives, or, Being First-Class is not Enough

Enno Scholz
email: scholz@inf.fu-berlin.de

B 95-1
January, 1995

Abstract. A monad is presented which is suitable for writing concurrent programs in a purely functional programming language. In contrast to, for instance, the IO monad [Launchbury, Peyton Jones 94], the primitives added to the functional language are not represented as built-in functions operating on the monad, but rather by Perry-style constructors [Perry 90] of a distinguished algebraic data type. Therefore, monadic expressions representing concurrent computations are not only first-class objects of the language; in addition, they may even be decomposed.
A number of examples show that decomposability of concurrent code is crucial for the purely functional construction of more powerful concurrency abstractions like rendezvous, remote procedure call, and critical regions from the primitives.
The paper argues that this technique helps to remedy a recurrent dilemma in the design of concurrent programming languages, namely, how to keep the language small, coherent, and rigorously defined, yet to provide the programmer with all the communication constructs required.
It is suggested that functional languages are not only capable of describing concurrent programs, but that in terms of expressiveness they may even prove to be superior to their imperative siblings.

1. Introduction

In order to describe the flow of control and data in concurrent programs, a number of different communication constructs like synchronous or asynchronous message-passing, rendezvous, and remote procedure call exist. For different applications, different communication constructs are adequate [Bal et al. 89]. It is well-known that rules can be given how to reformulate a program using one set of constructs in terms of another set [Andrews 91]. However, because such rules take program patterns as their parameters, in conventional programming languages they cannot be formulated by the programmer.

Traditionally, it is the designer of a concurrent programming language who is responsible for choosing the required communication constructs, since they must be built into the language. The language designer's dilemma is to either provide a rather limited set of communication primitives in order to keep the language small, which may severely restrict the language's application area, or to try and provide all the communication constructs found to be useful in concurrent programming, which may result in a complex language that is difficult to treat formally. An example for the former case is OCCAM [Inmos Ltd. 84], which provides a small number of rigorously defined constructs. Whilst inheriting a large body of theoretical properties from CSP [Hoare 85], its practical applicability is limited. An example for the latter case is SR [Andrews et al. 88], which provides a plethora of communication primitives but reflects their interrelatedness only in the language syntax.

An alternative explored in this paper is to place the responsibility (and the opportunity!) for designing the required communication constructs with the programmer. Only a minimal set of very simple communication primitives with a rigorously defined operational semantics is built into a functional language in such a way that concurrent programs composed of these primitives are not only first-class values of the language (which means they can be passed as parameters and stored in data structures), but additionally, they may be decomposed. We show that this decomposability is crucial for constructing more powerful concurrency mechanisms *within* the functional language, such that they derive their operational semantics from the primitives. This seems to be a solution for the above-mentioned dilemma and suggests that, in some respects, functional languages may be more expressive than imperative languages.

Decomposability of concurrency primitives is achieved by representing them as constructors of a distinguished algebraic data type in the style of Perry's *Result* type [Perry 90]. Compared to representing the primitives as built-in functions operating on a monad (as, for instance, in the *IO* monad [Launchbury, Peyton Jones 94]), composability comes for free: being functional data terms, the primitives can be decomposed using pattern-matching; however, as demonstrated in [Wadler 92], they can still be considered functions on a monad and be manipulated accordingly. The remainder of the paper is organized as follows: Section 2 explains the notation used; Section 3 gives a brief review of existing approaches to write concurrent programs using functional languages; Section 4 explains which concurrency primitives are to be added to the functional language and what their operational semantics is; Section 5 contains a concurrent example program in both continuation-passing style and monadic style; Section 6 sets up the framework for shifting between both styles; finally, Sections 7 through 10 take the reader through a series of example implementations of common concurrency mechanisms like rendezvous, remote procedure call, and critical regions.

2. Notation

The notation used in this paper is essentially the functional programming language HASKELL [Hudak et al. 92], which is a proposed standard and the functional programming community's preferred vehicle of scientific discourse. However, we assume the following extensions:

1. A type system with constructor classes and special syntax for monads as documented and implemented in the functional programming system GOFER, version 2.30 [Jones 94]. As far as these features are prerequisites for following our presentation, they are explained in Section 5.
2. The possibility to declare constructors of type synonyms as instances of constructor classes. There is an implementation restriction in Gofer saying this is only possible for type constructors of algebraic data types.
3. A facility for dynamic typing as documented and implemented in Yale Haskell, version 2.2 [Peterson 94]. It consists of a primitive abstract data type *Dynamic*, on which two functions *toDynamic* and *fromDynamic* are defined.

   $$\begin{aligned} toDynamic &:: a \rightarrow Dynamic \\ fromDynamic &:: Dynamic \rightarrow a \end{aligned}$$

   toDynamic succeeds for expressions of arbitrary type. It tags the expression with the type the compiler infers for type variable *a* in the context of the application of *toDynamic*. *fromDynamic* checks this tag against the inferred type for variable *a* in the context of its own application. If this type is no less general than the tag, *fromDynamic* returns the expression with the tag removed, otherwise it causes a runtime error.

To make the paper self-contained, an informal description of functions from the Haskell standard prelude is given where they appear in the example programs. For their definitions, refer to [Hudak et al. 92].

We have built an interleaving implementation of the concurrency monad presented here by extending Mark Jones's Gofer environment [Jones 94] to handle dynamic typing and the concurrency primitives. All programs presented in the course of this paper have been executed using this implementation.

3. Related Work

The evaluation of past attempts to use the functional paradigm as a basis for describing the interaction of systems of concurrent processes shows that this task is not easily accomplished. The most attractive point in functional programming, namely *equational reasoning*, fundamentally conflicts with the most prominent feature of concurrent programs, namely *nondeterminism* [Hughes, O'Donnell 92]. The most obvious approach is to model a process by a recursive function that maps a list of input messages to a list of output messages. This is called the *stream-processing approach*. However, this approach has several drawbacks. First, since the availability of the messages that a process receives on its input stream may depend on those it has output at a previous point in time, the programmer can easily produce deadlocks by trying to prematurely access elements of the input stream. Second, the resulting layout of communication channels between processes is unduly restricted, making the construction of client-server applications impractical. Third, this approach only works for deterministic programs. To enable the stream-processing approach to handle client-server interactions and nondeterminism, various schemes have been suggested. In all of [Henderson 82], [Broy 86], [Stoye 86], [Turner 87], [Darlington, While 87] [Jones, Sinclair 89], the functional language is extended with an additional primitive which enables nondeterministic functions to be constructed. To preserve some opportunities for equational reasoning, however, most of the authors propose schemes for restricting the usage of the nonfunctional primitive within the program.

The necessity of introducing *some* nonfunctional component into the system has led some researchers to the belief that in order to faithfully model concurrent systems with functional languages, equational reasoning has to be sacrificed completely. Thus, languages like Concurrent ML [Reppy 93] and Facile [Thomsen et al. 93] came into existence, which allow purely functional and side-effecting computations to be arbitrarily interspersed. Unfortunately, formal reasoning about the correctness of programs written in these languages is about as difficult as in imperative languages.

In order to preserve equational reasoning, an alternative approach is to embed a functional language in some kind of outer layer where nonfunctional computations are possible. Either a completely independent syntax is chosen for this purpose, like in [Lock, Jähnichen 90] and in the approaches based on process algebra (e.g., LOTOS [ISO 87], PSF [Mauw 91]), or the constructs of the outer layer are represented as a set of functions on a distinguished data type. A program with nonfunctional effects is represented as an object of this type; the evaluation of an object of this type may trigger the evaluation of purely functional expressions, but not vice versa. Depending on the laws holding for the distinguished type, the *continuation-passing style* (abbrev. *CPS*) *approach* [Karlsson 81], [Perry 90] and the *monadic approach* [Wadler 90], [Wadler 92] can be distinguished. The equivalence of the CPS approach and the monadic approach have been demonstrated in [Wadler 92] where the use of the latter style is advocated.

However, the proliferation of concurrency primitives is a problem shared by all approaches known to us. In [Scholz 95], we have proposed the decomposability of Perry-style constructor primitives as a possible solution; this work is extended here to cover the monadic framework.

4. The Concurrency Primitives

The concurrency primitives we introduce to extend the functional language are based on the paradigm of point-to-point, unidirectional, order-preserving, asynchronous, buffered message passing with explicit message receipt using an asymmetric naming scheme. I.e., sending a message is non-blocking and requires the sender to know the name of the receiver. A process wishing to receive a message explicitly issues an instruction which causes its execution to halt until a message from an unspecified sender (whose identity is possibly unknown to the receiver) has arrived. Messages sent to a process are buffered using an unbounded message queue. The order in which several messages were sent by process *p1* to process *p2* is preserved in *p2*'s message queue. Each process in the system is uniquely identified by a process identifier (abbrev. *PID*).

The operations that a process can perform are: send a message to another process (*Send*), wait for the arrival of a message (*Receive*), ask the operating system for the process's own PID (*OwnPid*), create a child process (*Fork*), and terminate (*End*).

Send takes the receiver process's PID and the data item to be transmitted as its parameters. Both *Receive* and *OwnPid* take no parameters. *OwnPid* immediately returns the current process's PID. In case the process's message queue is nonempty, *Receive* immediately returns the message at the head of the queue, otherwise it blocks until the queue is nonempty. *Fork* takes the code of the child process to be created as its parameter and returns the child process's PID. *End* takes no parameters.

4.1. Syntax of the Primitives

The above-mentioned instructions are represented as the constructors of a distinguished algebraic data type *Process* which is defined in Fig. 1. These constructors are called *process constructors*. The type *Process* is exactly analogous to the type *Result* in [Perry 90].

data Process	=	***Send*** *Pid Message (() → Process)*
	|	***Receive*** *(Message → Process)*
	|	***OwnPid*** *(Pid → Process)*
	|	***Fork*** *Process (Pid → Process)*
	|	***End***

Fig. 1: The example language's syntax

Each process constructor corresponds to one operation, taking one argument for each of the operation's arguments. With the exception of *End*, each of the constructors takes an additional argument representing the continuation process. The values returned by *Receive*, *Fork* and *OwnPid* are fed into the continuation process by means of one parameter.

Note that the data type *Message* is left to the programmer to be defined and extended according to the application's requirements. The data type *Pid*, however, is an abstract data type which has a representation that dependends on the language implementation. There are no operations on this data type visible to the programmer.

4.2. Operational Semantics of the Primitives

In Fig. 2, where the primitives' operational semantics is defined, the expression <> denotes the empty queue. *x^xs* and *xs^x* denote a queue *xs* with an element *x* added at the front, or at the end, respectively. The operator $\oplus$ denotes the union of two sets with empty intersection.

$ds \oplus \{ (pid, m\hat{}ms, \textbf{Receive}\ p)\} \Rightarrow ds \cup \{ (pid, ms, p\ m) \}$

$ds \oplus \{ (pid, ms, \textbf{OwnPid}\ p) \} \Rightarrow ds \cup \{ (pid, ms, p\ pid) \}$

$ds \oplus \{ (pid, ms, \textbf{End}) \} \Rightarrow ds$

$ds \oplus \{ (pid, ms, \textbf{Fork}\ p'\ p) \} \Rightarrow ds \cup \{ (pid', <>, p'), (pid, ms, p\ pid') \}$

$\quad where\ \ pid' \neq pid \wedge \forall\ (pid'',x,y) \in ds: pid' \neq pid''$

$ds \oplus \{(pid, ms, \textbf{Send}\ pid'\ m\ p)\} \Rightarrow \{f\ d \mid d \in ds \cup \{(pid, ms, p\ ())\}\}$

$\quad where\ \ f\ (pid'',ms,p') = (pid'', ms\hat{}m, p') \quad if\ \ pid''= pid'$

$\qquad\qquad\qquad\qquad\qquad (pid'', ms, p') \quad otherwise$

Fig. 2: The example language's operational semantics

The operational semantics of the primitives is given by a nondeterministic transition relation on states of the world. In order to capture the behaviour of processes, we model the world as a set of process descriptors *ds*. Each process descriptor describes a point in the execution of one process. A process descriptor has three entries, namely, the process's PID *pid*, its message queue *ms*, and a term of type *Process* representing the code which remains to be executed.

Similar to the execution of a functional program, which is initiated by specifying a top-level expression to be evaluated, the execution of a concurrent program is started by specifying a data object of type *Process*. To run a process *p* with an arbitrary PID *pid*, a world is created with an initial state that contains the process descriptor *(pid, <>, p)* as its only element. Repeatedly, from the rules given in Fig. 2, one that matches the current state of the world is selected nondeterministically and applied to the current state of the world, yielding a new state. This procedure is repeated until the world reaches a state such that there is no matching rule.

We have now completed the definition of the concurrency primitives' syntax and operational semantics. Obviously, the underlying communication paradigm is of utmost simplicity. In the sequel, we show that that the primitives provided are not merely suitable for the construction of serious programs, but can indeed serve as a building-blocks for customized communication mechanisms which are considerably more powerful.

5. An Example Program

In this section, we present an example program in CPS form. We then introduce the monad *P* that corresponds to the type *Process*. We show how each object of type *Process* can be transformed into an equivalent object of type *P ()*, and vice versa. Finally, the example program is reformulated in monadic style using the special syntax for monads.

5.1. CPS Version

The first step in constructing a collection of concurrent processes is to define the types of messages they use to communicate. Here we need messages containing either a list of integer values or a single integer value.

```
data Message =  ..  |  IntList [Int]  |  Int Int
```

The process *addUp* receives a list of numbers (wrapped in constructor *IntList*) from a client and returns their sum (wrapped in constructor *Int*) to the client. *addUp* takes as a parameter the PID of its client, i.e., the process which is to receive the result of its computation.

In the following code for *addUp*, note that the construct *\v → e* is Haskell syntax for the lambda abstraction $\lambda v.e$. The operator *($)* and the functions *splitAt* and *length* are defined in the standard prelude. The operator *($)* denotes function application. The only reason for using *($)* is that it saves a lot of parentheses: in Haskell, infix operators have lower precedence than prefix operators, thus we can write *f $ g $ h $ j x* instead of *f (g (h (j x)))* and *f $ \x → g x* instead of *f (\x → g x)*. The function *splitAt n* breaks a list at element *n*, and *length* calculates the length of a list.

```
addUp :: Pid → Process
addUp client =
  Receive $ \(ListInt ns) →
     case ns of
     [n]→  Send client (Int n) $ \() →
           End
     _  →  OwnPid $ \self →
           let (ns1,ns2) = splitAt (length ns / 2) ns in
           Fork (addUp self) $ \server1 →
           Fork (addUp self) $ \server2 →
           Send server1 (ListInt ns1) $ \() →
           Send server2 (ListInt ns2) $ \() →
           Receive $ \(Int n1) →
           Receive $ \(Int n2) →
           Send client (Int (n1 + n2)) $ \() →
           End
```

Initially, *addUp* waits for the arrival of the list of numbers to be summed up. In case this list consists of one number only, this number is returned to the client. Otherwise, it is split in two halves of approximately equal length. The current process then creates two additional *addUp* processes, which are supplied with the current process's PID, i.e., the current process acts as their client. The two halves are sent to the child processes. The current process waits for their results, then returns their sum to its client and terminates.

This process uses *addUp* to compute the sum of the list *1,2,..,20*:

```
addUpMain :: Process
addUpMain =
  OwnPid $ \self →
  Fork (addUp self) $ \server →
  Send server (ListInt [1..20]) $ \() →
  Receive $ \(Int n) →
  End
```

The CPS version of the *addUp* program has two flaws: its syntax is clumsy, and a *Process* cannot return a value. This will be remedied shortly.

5.2. Process Continuations are Monads

Although a *Process* cannot return a value, it can apply a continuation process to a value.

This programming technique is illustrated by a CPS factorial function, which has an (admittedly rather contrived) side effect, namely, forking an arbitrary process named *px*.

```
fac :: Int → (Int → Process) → Process
fac 0 c   =  Fork px $ \_ →
               c 1
fac n c   =  fac (n - 1) $ \res →
               c (n * res)
```

In addition to the number *n* for which the factorial is to be computed, *fac* takes a continuation process *c* which is to be applied to the result of this computation. In case *n = 0*, the result is *1*; process *px* is forked and the *fac* process continues by applying the process continuation to *1*. In case *n ≥ 0*, *fac (n - 1)* is executed. However, it cannot be passed the original continuation since this expects to be applied to the result of *fac n*, not to the result of *fac (n - 1)*. The correct continuation for *fac (n - 1)* multiplies the result of *fac (n - 1)* with *n* and applies the original continuation *c* to it.

Obviously, the CPS equivalent of a function returning an object of type *a* is a function returning an object of type *(a → Process) → Process*. We therefore introduce a type synonym *P*.

```
type P a = (a → Process) → Process
```

Following [Wadler 92], we can consider *P* a monad by defining two functions *result* and *bind*. In Gofer, this can be done by declaring *P* to be an instance of the constructor class *Monad*.

```
instance Monad P where result a   =  \c → c a
                       bind pa f  =  \c → pa (\a → f a c)
```

This is the simplified interface of constructor class *Monad*.

```
class Monad m  where  result :: a → m a
                      bind :: m a → (a → m b) → m b
```

Now, functions passing *Process* continuations can be rewritten in monadic style. Especially, note that the concurrency primitives, except for *End*, can be considered functions on the *P* monad. The type of *Fork*, for instance, is *Process → P Pid*; *Send*'s type is *Pid → Message → P ()*. Functions with result type *P ()* are called *commands*.

Here is the monadic version of *fac*. Note that, in Haskell, a function identifier enclosed in backquotes serves as an infix operator.

```
fac :: Int → P Int
fac 0 =  Fork px `bind` \_ →
         result 1
fac n =  fac (n - 1) `bind` \res →
         result (n * res)
```

Using the special syntax for monads suggested by Mark Jones and implemented in the Gofer system, this can be written more legibly:

```
fac :: Int → P Int
fac 0 =  do  Fork px
             [1]
fac n =  do  res ← fac (n - 1)
             [n * res]
```

In general, this syntax is defined as follows: an expression of type *m a*, where *m* is a monad type constructor, is started by keyword *do* followed by a nonempty list of entries, of which the last must be of type *m a* and is called *tail expression*. The others are called *qualifiers*. The rules for turning a *do* expression into one using *`bind`* are given in Fig. 3.

do { *Pat ← Exp***;** *Rest* **}**	⇒	*Exp* **`bind`** *\Pat →* **do {** *Rest* **}**
do { *Exp***;** *Rest* **}**	⇒	*Exp* **`bind`** *_ →* **do {** *Rest* **}**
do { let { *..* **};** *Rest* **}**	⇒	**let {** *..* **} in** **do {** *Rest* **}**
do { *Exp* **}**	⇒	*Exp*

Fig 3: Special syntax for monads

Note that, in contrast to a qualifier, a tail expression is not changed. Furthermore, the equivalence of *[x]* and *result x* in the list monad is adopted to hold for arbitrary monads in this syntax.

Monadic expressions of type *P ()* are equivalent to "raw" processes. One can be transformed into the other by means of *toP* and *fromP*.

```
toP :: Process → P ()
toP End                =  [()]
toP (Receive p)        =  do  m ← Receive;      toP (p m)
toP (Send pid m p)     =  do  Send pid m;       toP (p ())
toP (OwnPid p)         =  do  self ← OwnPid;    toP (p self)
toP (Fork p' p)        =  do  pid ← Fork p';    toP (p pid)

fromP :: P () → Process
fromP f = f (\() → End)
```

Because, in general, we are going to think in terms of monadic functions, and not in terms of processes, we need a version *fork* of the *Fork* constructor that takes an object of type *P ()* as the parameter it is going to fork.

```
fork :: P () → P Pid
fork p = Fork (fromP p)
```

5.3. Monadic Version

The example program *addUp* which was previously presented in CPS syntax can now be rewritten in monadic syntax.

```
addUp :: Pid → P ()
addUp client =
  do  ListInt ns ← Receive
      case ns of
     [n]  →  do  Send client (Int n)
     _    →  do  self ← OwnPid
                 let (ns1,ns2) = splitAt (length ns / 2) ns
                 server1 ← fork (addUp self)
                 server2 ← fork (addUp self)
                 Send server1 (ListInt ns1)
                 Send server2 (ListInt ns2)
                 Int n1 ← Receive
                 Int n2 ← Receive
                 Send client (Int (n1 + n2))
```

addUpMain can now be written as a function on the *P* monad which returns the sum of the elements of its list-valued argument.

```
addUpMain :: [Int] → P Int
addUpMain xs =
  do  self ← OwnPid
      server ← fork (addUp self)
      Send server xs
      Int n ← Receive
      [n]
```

6. Modifiers

Until now, it is not obvious why we take the trouble of defining the *Process* data type and its constructor functions *Send*, *Receive*, *OwnPid*, *Fork*, and *End* as primitives, instead of providing the monad *P* and a set of (non-constructor) functions *send*, *receive*, *ownPid*, and *fork* as primitives. The reason is that, this way, monadic expressions can be decomposed by functional means.

We will use this property to extend our power of expression beyond the narrow scope of the built-in primitives, defining so-called *modifiers*, which decompose a piece of code and rebuild it in a modified way. As was demonstrated in the previous section, a command may be manipulated either as an object of type *P ()* or as an object of type *Process*. For composing pieces of code, type *P ()* is more pleasant, since monadic syntax can be used. For decomposition, however, objects of an algebraic data type are required, i.e., they must have type *Process*. This is why modifiers have the following type.

```
type Modifier = Process → P ()
```

Command *modify* applies a modifier to the current process's continuation.

```
modify :: Modifier → P ()
modify f = \p → fromP (f (p ()))
```

A simple, yet useful, example for a modifier is *delay*.

```
delay :: P () → Modifier
delay c End              = [()]
delay c (Send pid m p)  = do Send pid m;      c; toP (p ())
delay c (Receive p)     = do m ← Receive;    c; toP (p m)
delay c (OwnPid p)      = do pid ← OwnPid;c; toP (p pid)
delay c (Fork p' p)     = do pid ← Fork p'; c; toP(p pid)
```

As is the case with all modifiers, the effect of *delay* can either be explained from a static perspective or from a dynamic perspective. From a static perspective, *modify (delay c)* inserts a command *c* into the remaining code at the position after the next command, if this exists. Thus, in a context where *m* and *pid* are defined,

```
do  modify (delay (Send pid m))
    self ← OwnPid
    Send self m
```

is equivalent to

```
do  self ← OwnPid
    Send pid m
    Send self m
```

From a dynamic perspective, *modify (delay com)* delays the execution of *com* until after the next instruction. As the remainder of the paper will show, the most natural perspective to the user of a modifier is, in general, the dynamic perspective.

7. Guarded Receive

The communication primitives built into our example language are rather simple-minded. For instance, using *Receive*, a process must process all messages in the order of their arrival. For many applications, this is fine. In others, however, a process may need to wait for the arrival of a specific message in order to be able to process other messages which possibly arrived earlier.

In principle, this behaviour can be implemented by successively executing *Receive* and storing the messages received in a temporary buffer until the arrival of the desired message. All subsequent applications of *Receive* must then be replaced by instructions that take messages from the temporary buffer until this is empty. Unfortunately, if the concurrency primitives like *Receive* are merely first-class, but not decomposable, there is no way to do this. Hence, the well-known interrelatedness of the various communication paradigms cannot be exploited: a new communication primitive must be provided, or the programmer has to think of an ad-hoc work-around. With decomposable primitives, though, the replacement of *Receive* statements can be accomplished by a suitable modifier.

In this section, we will define a more comfortable receive operation which allows the user to specify a guard, i.e., a predicate that is required to hold for the message received.

7.1. Interface

The operation *guardedReceive* takes a predicate *guard* on messages as a parameter.

```
guardedReceive :: (Message → Bool) → P Message
```

guardedReceive buffers all incoming messages until the arrival of a message *m* for which its argument predicate *guard* evaluates to *True*. Then, it puts the buffered messages back into the queue, preserving their order, and returns *m*.

7.2. Implementation

As a first step, we define a modifier *pushQueue*. From a dynamic perspective, *modify (pushQueue m)* places a message *m* at the beginning of the current process's message queue.

```
pushQueue :: Message → Modifier
pushQueue m (Receive p) =
   do  toP (p m)
pushQueue m other =
   do  modify (delay (modify (pushQueue m)))
       toP other
```

This effect is produced by recursively traversing the current process's continuation until the first *Receive* operation is found. Message *m* is then fed into the first *Receive* operation's continuation.

Using modifier *pushQueue, guardedReceive* can be defined.

```
guardedReceive :: (Message → Bool) → P Message
guardedReceive guard =
   do  m ← Receive
       if guard m then
            [m]
        else
            do  m' ← guardedReceive guard
                modify (pushQueue m)
                [m']
```

The correctness of *guardedReceive* is best seen by induction over the position *n* of the first element in the message queue for which the predicate *guard* holds. For *n = 1*, *guardedReceive* returns immediately. Otherwise, given that the message queue is in the right order after execution of *m'* ← *guardedReceive guard*, the only thing left to do to ensure that the message queue will be in the correct order after termination is to use *pushQueue* to place the message *m*, which arrived first of all, at the beginning of the queue.

8. Rendezvous

As mentioned in Section 4, the naming scheme implemented by the built-in primitives is asymmetric, i.e., a receiver is not able to specify the sender's identity. We will now present a mechanism for symmetric communication called *rendezvous*.

8.1. Interface

The symmetric counterparts to *Send* and *Receive* are called *put* and *get*.

```
put :: Pid → Message → P ()
get :: Pid → P Message
```

put has the same signature and (to the user) the same behaviour as *Send*. *get* takes the sender's PID as an extra argument. In contrast to *Receive*, it does not return the first message in the message queue, but buffers all incoming messages until the arrival of a message *m* from the chosen sender. Then, it puts the buffered messages back into the queue, preserving their order, and returns *m*.

8.2. Implementation

A symmetric naming scheme can easily be implemented on top of an asymmetric one: each message is tagged with the sending process's PID. To that end, a message constructor *From* is introduced.

```
data Message = .. | From Pid Message
```

put tags the message being sent with the sender's PID:

```
put :: Pid → Message → P ()
put pid' m =
   do self ← OwnPid
      Send pid' (From self m)
```

get uses *guardedReceive* to wait for the arrival of a message which is wrapped in a *From* constructor tagged with the sender's PID.

```
get :: Pid → P Message
get pid =
   do From _ m ← guardedReceive guard
      [m]
   where
   guard (From pid' _) | pid' == pid   = True
   guard _                             = False
```

8.3. Application

Using these rendezvous constructs, we present a generic concurrent divide-and-conquer process *divAndConq*. It waits for a problem contained in a message with constructor *Unsolved* and returns a solution wrapped in constructor *Solved* to its client:

```
data Message = .. | Unsolved Problem | Solved Solution
```

divAndConq takes its client's PID and a quadruple of functions (named *isTrivial*, *trivial*, *divide* and *merge*) as its arguments. On receiving a problem *p*, it tests whether it is trivial (using *isTrivial*). In this case, the solution (obtained by applying *trivial*) is returned to the client immediately. Otherwise, the problem is divided into two subproblems (using *divide*), which are solved by two child processes. The corresponding subsolutions are composed (using *merge*) and returned to the client.

```
divAndConq::(Problem → Bool,
             Problem → Solution,
             Problem → (Problem,Problem),
             Solution → Solution → Solution) → Pid → P ()
divAndConq (fs @(isTrivial,trivial,divide,merge)) client =
  do Unsolved p ← get client
     if not (isTrivial p) then
        do let (p1,p2) = divide p
           self ← OwnPid
           child1 ← fork (divAndConq fs) self
           child2 ← fork (divAndConq fs) self
           put child1 (Unsolved p1)
           put child2 (Unsolved p2)
           Solved s1 ← get child1
           Solved s2 ← get child2
           put client (Solved (merge s1 s2))
     else
        do put client (Solved (trivial p))
```

Note that the use of *get* in *divAndConq* ensures that the answers from *child1* and *child2* are processed in this order, independently of the order of their arrival. This is crucial for the algorithm's correctness.

Taking lists of integers to be the problem and solution domain, the following use of *divAndConq* yields a popular sorting algorithm.

```
type Problem = [Int]
type Solution = [Int]

quickSort :: [Int] → P [Int]
quickSort p =
  do self ←OwnPid
     child ← fork (divAndConq(isTrivial,id,divide,(++)) self)
     put child (Unsolved p)
     Solved s ← get child
     [s]
  where
  isTrivial       =  (<=1) . length
  divide (x:xs)   =  let as = [ x' | x' ← xs, x' ≤  x ]
                         bs = [ x' | x' ← xs, x > x ] in
                     if null as then ([x], bs) else (as, x:bs)
```

The standard prelude functions *head*, *null*, and *(++)* return the first element of a list, test a list for emptiness, and concatenate two list, respectively.

8.4. Remarks

This is a simple example of how different layers in a message-passing protocol can be isolated against each other. For instance, processes communicating via *get* and *put* do not need to know how the layer that tags and untags messages with information about the sending process is implemented.

Unfortunately, Haskell's type system does not provide any means to extend the (public) data type *Message* with additional (private) constructors. Therefore, the rendezvous mechanism cannot be made completely opaque.

9. Remote Procedure Call

The next communication mechanism we wish to model is the *remote procedure call*. Given the purely functional definition of an abstract data type (abbrev. *ADT*), we provide a mechanism for defining multiple server processes, each with a unique identity, offering the ADT's operations as remote procedures to arbitrary client processes. This is accomplished without recoding the ADT's interface in terms of concurrency constructs (the importance of avoiding this was pointed out, with reference to ADA, by [Andrews 88]).

9.1. Interface

The interface of the remote procedure mechanism consists of two operations *spawn* and *(?)*.

```
spawn :: a → P (Ref a)
(?) :: Ref a → (a → (b,a)) → P b
```

The function *spawn* takes an object, creates a server for it, and returns a reference to the server. The function *(?)* takes as its parameters a reference *ref* to a server for an object of type *a* and a state transformer *f* on type *a* returning an object of type *b*. The command *ref ? f* causes *f* to be sent to the server process, which applies *f* to its state, updates its state accordingly, and sends an object of type *b* back to the client.

9.2. Implementation

The client and the server communicate via dynamically typed objects. In the sequel, *wrapping* an object means applying *toDynamic* to it and *unwrapping* it means applying *fromDynamic*.

```
data Message = .. | Dynamic Dynamic
```

To ensure that only state transformers of the appropriate type are sent to a server, a reference to a server for an object of type *a*, which is implemented by the server's PID, is associated with *a*'s type. In Gofer, the following definition of the reference type *Ref* does the trick:

```
data Ref a =Ref Pid          -- Gofer
```

Note that in Haskell, we would have to make a definition

```
data Ref a = Ref Pid a       -- Haskell
```

which forces us to always pass a dummy parameter around at run-time in order to get the types right at compile-time.

This ensures that an attempt to invoke a procedure on a server for an object of a non-matching type is rejected by the compiler as a type error. Only the implementation, which is correct, makes use of the type-unsafe features. Since the interface of the remote procedure call mechanism is completely typesafe, no run-time errors can occur.

spawn a is implemented by forking a server process for *a* and returning a typed reference to the server process, which is done by wrapping its PID in constructor *Ref*.

```
spawn :: a → P (Ref a)
spawn a =
  do pid ← fork (server a)
     [Ref pid]
```

A server process *server a* for an object *a* waits for a message containing the client's PID and a (dynamically typed) state transformer *f*. On the arrival of a request, *f* is unwrapped and applied to *a*. The state transformer's result value *b* is wrapped and sent to the client. The server then updates its state and resumes waiting.

```
server :: a → P ()
server a =
  do From client (Dynamic f) ← Receive
     let (b, a') = (fromDynamic f) a
     put client (Dynamic (toDynamic b))
     server a'
```

Invoking a remote procedure using *(?)* consists of wrapping the state transformer *f*, sending it to the server process designated by the PID stored in the server reference, waiting for the server's reply *b*, unwrapping it, and returning it.

```
(?) :: Ref a → (a → (b,a)) → P b
(Ref server) ? f =
  do put server (Dynamic (toDynamic f))
     Dynamic m ← get server
     [fromDynamic m]
```

9.3. Application

Consider the following ADT *Dictionary* which offers dictionary services. Its interface consists of one generator function *createDictionary* and four operations *add*, *set*, *delete*, and *lookUp*. (Their definitions are omitted here.)

```
createDictionary :: [(String, Int)] → Dictionary

set      :: String → Int → Dictionary → ((), Dictionary)
add      :: String → Int → Dictionary → ((), Dictionary)
delete   :: String →        Dictionary → ((), Dictionary)
lookUp   :: String →        Dictionary → (Int, Dictionary)
```

The following process *rpcClient* illustrates the creation and use of an RPC server for objects of type *Dictionary*.

```
rpcClient :: P ()
rpcClient =
  do let dict1 = createDictionary [ ("Peter", 10000) ]
         dict2 = createDictionary [ ("Paul", 300),
                                    ("Mary", 850) ]
     richPeople ← spawn dict1
     poorPeople ← spawn dict2
     balance ← poorPeople?lookUp "Mary"
     poorPeople?delete "Mary"
     richPeople?add "Mary" (balance + 2000)
```

9.4. Remarks

We have assumed that the functions defining an ADT *a* were defined as state transformers (i.e., functions having result type $a \to (b, a)$ for arbitrary *b*). However, most functions either do not return a value or they do not alter the argument. In these cases, some glue code is needed which adds one dummy entry to the function's result.

10. Critical regions

Our last and largest example is a language mechanism called *critical regions*. Given a set of processes in which parts of each one's code are marked as critical, the mechanism ensures that, at any time, at most one of the processes executes critical code.

10.1. Interface

The interface of the critical regions mechanism consists of a command *CR* and a command *crRun*.

```
CR :: Mode → P ()
crRun :: [P ()] → P [Pid]
```

CR takes a parameter of type *Mode*.

```
data Mode = On | Off
```

CR On and CR *Off* mark the beginning and the end of a critical region, respectively. Note that a critical region cannot be delimited by only one command with type *P () → P ()*, since this would imply that variables bound within a critical region could not be used outside it. *crRun* takes as its argument a list of commands which are to be executed concurrently such that it is guaranteed that only one at a time can be in a critical region.

10.2. Implementation

Mutual exclusion is ensured using a *token ring algorithm*. All the processes are arranged in a virtual ring such that every process only knows the PIDs of its predecessor and its successor. A special message, called the *token*, circulates between the processes. Only the process holding the token may execute critical code. If a process terminates, it sends messages to its predecessor and its successor, telling them that they are now connected.

The types of messages required for the implementation of this algorithm are

```
data Message = .. | Token | NewPred Pid | NewSucc Pid
```

CR is implemented by extending the *Process* data type.

```
data Process = .. | CR Mode (() → Process)
```

However, note that the number of primitives is not augmented: *CR* has no associated primitive. Executing it causes a runtime error. *CR On* and *CR Off* only serve as a markers which are interpreted and removed by a modifier *crMod*.

From a static perspective, *crMod* is used to transform processes containing occurrences of command *CR* into processes that are suitable to be started and arranged in a virtual ring by *crRun*. From a dynamic perspective, a process that is part of a token ring carries additional state information which is managed by *crMod*. This state information consists of a parameter *sm* of type *Mode*, which indicates whether the process is currently within a critical region, the PID *sp* of the current process's predecessor in the token ring, and the PID *ss* of its successor.

Since the code for *crMod* is slightly longer than that of the examples encountered so far, we have divided it into numbered sections to make it easier for the reader to match code and explanatory remarks.

```
[1] crMod :: Mode → Pid → Pid → Modifier
    crMod Off sp ss End =
      do crMod Off sp ss (CR On (\() → End))
    crMod On sp ss End =
      do Send sp (NewSucc ss)
         Send ss (NewPred sp)
         Send ss Token
[2] crMod Off sp ss (CR On p) =
      do m ← guardedReceive (\m→ or[isToken m,
                                    isNewPred m,
                                    isNewSucc m])
         case m of
           Token        → do crMod On sp ss (p ())
           NewPred sp'→ do crMod Off sp' ss (CR On p)
           NewSucc ss'→ do crMod Off sp ss' (CR On p)
[3] crMod On  sp ss (CR Off p) =
      do Send ss Token
         crMod Off sp ss (p ())
[4] crMod sm sp ss (CR sm' p) =
      do crMod sm sp ss (p ())
[5] crMod sm sp ss (Receive p) =
      do m ← Receive
         case m of
           Token        → do Send ss Token
                             crMod sm sp ss (Receive p)
           NewPred sp'→ do crMod sm sp' ss (Receive p)
           NewSucc ss'→ do crMod sm sp ss' (Receive p)
           _            → do crMod sm sp ss (p m)
[6] crMod sm sp ss other =
      do modify (delay (modify (crMod sm sp ss)))
         toP other
```

The modifier *crMod* gives special attention to constructors *End*, *CR* and *Receive*.

[1] In order to terminate, a process must first get hold of the token. In case several processes want to terminate simultaneously, this precondition ensures the correct reconfiguration of the token ring. On the arrival of the token, the process sends messages to its predecessor and its successor, introducing them to each other and connecting them. Its final action is to pass the token on to its successor.

[2] To begin a critical region, a process must wait for the arrival of the token. However, correct reconfiguration requires that all *NewPred* or *NewSucc* messages that arrive before the token are processed before the token is passed on. The predicates *isToken*, *isNewPred* and *isNewSucc* are assumed to hold exactly for messages with constructor *Token*, *NewPred*, and *NewSucc*, respectively. The standard prelude function *or* maps a list of booleans to *True* if at least one of them is *True*.

[3] If a process leaves a critical region, it must pass on the token.

[4] Within a critical region, occurrences of *CR On* are ignored, likewise occurrences of *CR Off* outside a critical region.

[5] Every occurrence of *Receive* in a mutex process has to be guarded against the arrival of one of the "system" messages *SetPred*, *SetSucc*, and *Token*. The unexpected arrival of the token can only happen while the process is outside a critical region; the token is then passed on. Messages *SetPred* and *SetSucc* are processed as above by updating the process's state.

[6] All other operations are ignored by *crMod*.

The function *crRun* starts the list of processes for which the mutual exclusion of critical regions is to be ensured. It uses *crMod* to handle occurences of *CR On* and *CR Off* in the processes and to initialize their state.

```
crRun :: [P ()] → P [Pid]
crRun ps =
  do pids ← parallel [ do modify (crMod Off undefined
                                            undefined)
                        p | p ← ps]
     let sp_pids = tail pids ++ [head pids]
         ss_pids = [last pids] ++ init pids
     parallel [ do Send pid (NewPred sp)
                   Send pid (NewSucc ss) |
         (pid, sp, ss) ← zip3 pids sp_pids ss_pids ]
     Send (head pids) Token
     [pids]
     where
     parallel :: [P()] → P [Pid]
     parallel []      =  [[]]
     parallel (p:ps)  =  do pid ← fork p
                            pids ← parallel ps
                            [pid:pids]
```

Initially, none of the processes is within a critical region, and the PIDs of their predecessors and successors are invalid. Each one waits for a message from *crRun* telling it the PIDs of its predecessor and its successor. Finally, *crRun* hands the token to the first process.

Note that the standard prelude functions *head* and *last* return the leftmost and the rightmost element of a list, respectively, while *tail* and *init* remove them. *undefined* is the standard prelude function representing ⊥. *zip3* merges three lists into a list of triples.

10.3. Application

The application *mutexMain* spawns a server for a *Dictionary* object with keys *"Peter"* and *"Paul"*. A reference to this server is passed to two worker processes which are started with *crRun*. Each of these worker processes expects an amount with which to credit *Paul's* account. In order to make this read-and-update operation atomic, the critical region construct is employed.

```
mutexMain :: P ()
mutexMain =
  do let dict = createDictionary [ ("Peter", 10000),
                                   ("Paul",  11000) ]
     dataBase ← spawn dict
     [pid1,pid2] ← crRun [ mutexWorker dataBase,
                           mutexWorker dataBase ]
     Send pid1(Int 100)
     Send pid2 (Int (- 100))

mutexWorker :: Ref Dictionary → P ()
mutexWorker dataBase =
  do Int amount ← Receive
     CR On
     balance ← dataBase?lookUp "Paul"
     dataBase?set "Paul" (balance + amount)
     CR Off
```

10.4. Remarks

Note that, in Haskell, the introduction of an additional constructor *CR* makes it necessary to add one line to the definitions of modifiers *toP* and *delay*, breaking their encapsulation. This is not necessary, however, for modifiers implemented on top of *toP* and *delay*. Moreover, note that the programmer is responsible for always using *crRun* to start processes containing occurrences of *CR*. Haskell's type system is not strong enough to enforce this. In a type system where a supertype *Process'* of *Process* could be defined, which extends *Process* with constructor *CR*, the typechecker could type processes containing occurrences *CR* with type *Process'* to prevent errors.

To end on a positive note, note that the application code and the token ring algorithm are completely separated: processes using *CR On* and *CR Off* are neither required to cope with the extra state information needed to ensure mutual exclusion, nor do they have to handle the messages used for synchronisation and reconfiguration.

11. Conclusions

We have presented a technique for writing concurrent programs in a purely functional programming language. Concurrency primitives were *introduced* in continuation-passing style but *manipulated* mainly in monadic style. By representing the concurrency primitives as constructor functions we made processes decomposable, which is a significantly more powerful property than simply being first-class.

By means of a series of examples of increasing complexity we have demonstrated that having decomposable processes significantly enhances the possibilities for reusing and refining existing concurrent code. Powerful communication mechanisms can be defined entirely within the functional framework, deriving a rigorously defined operational semantics from the built-in primitives. Thus, there is no need to introduce new primitives to accomodate specialized application demands. Instead, the fact that many communication constructs are just variations of each other can to a large degree be exploited by the programmer. Moreover, code designed to be used in a sequential program can be reused at the concurrency level with a minimum of glue code.

We have devoted considerable attention to the pragmatics of our technique. The point was not to demonstrate that coding concurrent programs using a functional language is possible, but that it is elegant and worth-while. It seems that, shifting between the continuation-passing style perspective and the monadic perspective according to need, common concurrent programming idioms can be expressed in a natural way and in a pleasant and concise syntax that avoids the awkwardness of existing stream-based and continuation-based techniques.

We have shown that different layers of a computer's concurrent software, which are traditionally implemented in different languages either at the operating system level, or at the compiler level, or at the program level, can all be constructed using one formalism, namely, a functional language.

Acknowledgements

I wish to thank Manuel Chakravarty, whose diligent proof-reading and insightful comments at various stages of this paper were extremely helpful. Thanks to Mark Jones for providing information about the Gofer implementation.

References

[Andrews et al. 88] G.R. Andrews et al.: *An Overview of the SR Language and Implementation*, ACM Transactions on Programming Languages and Systems, Vol. 10, No.1, 1988

[Andrews 91] G.R. Andrews: *Concurrent Programming - Principles and Practice*, Benjamin/Cummings, 1991

[Bal et al. 89] H.E. Bal, J.G. Steiner, A.S. Tanenbaum: *Programming Languages for Distributed Computing Systems*, ACM Computing Surveys, Vol. 21, No. 3, Sept. 1989

[Broy 86] M. Broy: *A Theory for Nondeterminism, Parallelism, Communication, and Concurrency*, Theoretical Computer Science 45, pp 1-61, 1986

[Darlington, While 87] J. Darlington, L. While: *Controlling the Behaviour of Functional Language Systems*, Conference on Functional Programming and Computer Architectures, Portland, Oregon, 1987

[Henderson 82] P. Henderson: *Purely Functional Operating Systems*, in J. Darlington, P. Henderson, D. Turner: *Functional Programming and its Application: An Advanced Course*, Cambridge University Press, 1982

[Hoare 85] C.A.R. Hoare: *Communicating Sequential Processes*, Prentice-Hall, 1985

[Hudak et al. 92] P. Hudak, S. Peyton Jones, P. Wadler (editors): *Report on the Programming Language Haskell: Version 1.1*, ACM SIGPLAN Notices, 27 (5), May 1992

[Hughes, O'Donnell 92] J. Hughes, J. O'Donnell: *Expressing and Reasoning About Non-Deterministic Functional Programs*, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, 1992

[Inmos Ltd. 84] Inmos Ltd.: *Occam Programming Manual*, 1984 Prentice-Hall International

[ISO 87] *LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observable Behaviour*, International Organization for Standardization, 1987

[Jones, Sinclair 89] S.B. Jones, A.F. Sinclair: *Functional Programming and Operating Systems* The Computer Journal Vol. 32 (2), pp 162-174, 1989

[Jones 94] M. Jones: *Gofer 2.21/2.28/2.30 Release Notes*, available by anonymous ftp from `ftp.cs.yale.edu`

[Karlsson 81] K. Karlsson: *Nebula, a Functional Operating System*, Chalmers University, Göteborg, 1981

[Launchbury, Peyton Jones 94] J. Launchbury, S.L. Peyton Jones: *Lazy Functional State Threads*, Conference on Programming Language Design and Implementation, Orlando, FL, June 1994

[Lock, Jähnichen 90] H.C.R. Lock, S. Jähnichen: *Linda Meets Functional Programming*, Proc. of 2nd IEEE Workshop on Future Trends in Distributed Computing Systems, 1990

[Mauw 91] S. Mauw: *PSF, A Process Specification Formalism*, PhD Thesis, University of Amsterdam, 1991

[Peterson 94] J. Peterson: *Dynamic Typing in Haskell*, Research Report YALEU/DCS/RR-1022, Yale University, 1994

[Perry 90] N. Perry: *The Implementation of Practical Functional Programming Languages*, PhD thesis, Imperial College, University of London, 1990

[Reppy 93] J.H. Reppy: *Concurrent Programming with Events - The Concurrent ML Manual*, AT & T Bell Laboratories, 1993

[Scholz 95] E. Scholz: *Turning a Functional Data Type into a Concurrent Programming Language*, 9th ACM SIGAPP Symposium on Applied Computing, Nashville, Tennessee, 1995 (to appear)

[Stoye 86] W. Stoye: *Message-based Functional Operating Systems*, Science of Computer Programming Vol. 6, pp 291-311, 1986

[Thomsen et al. 93] B. Thomsen et al: *Facile Antigua Release Programming Guide*, Technical Report ECRC-93-20, 1993 European Computer-Industry Research Centre

[Turner 87] D. Turner: *Functional Programming and Communicating Processes*, Conference on Parallel Languages and Architectures, 1987

[Wadler 90] P. Wadler: *Comprehending Monads*, ACM Conference on Lisp and Functional Programming, Nice, France, June 1990

[Wadler 92] P. Wadler: *The Essence of Functional Programming*, 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1992