

# Four Concurrency Primitives for Haskell

Enno Scholz\*

Freie Universität Berlin  
Institut für Informatik  
Takustr. 9, 14195 Berlin

Email: scholz@inf.fu-berlin.de  
Fax: +49-30-83875109

**Abstract.** A monad for concurrent programming that is suitable for being built into Haskell is presented. The monad consists of only four primitives with a very simple semantics. A number of examples demonstrate that monads encapsulating other, more sophisticated communication paradigms known from concurrent functional languages such as Concurrent ML, Facile, and Erlang can be naturally and systematically constructed from the built-in monad in a purely functional way.

The paper argues that minimizing the number and complexity of the concurrency primitives and maximizing the use of purely functional abstractions in the design of concurrent languages helps to remedy a recurrent dilemma, namely, how to keep the language small and rigorously defined, yet to provide the programmer with all the communication constructs required.

An interleaving implementation of the monad has been built by extending Mark Jones's Gofer environment to handle the concurrency primitives. All programs presented in the course of the paper have been executed using this implementation.

## 1. Introduction

The topic of concurrency has recently received much attention in programming language research, in general, and in functional programming research, in particular. There seems to be a fundamental conflict between the inherent nondeterminism of concurrent computations and the referential transparency that is considered to be the main virtue of functional languages. Thus, languages like Concurrent ML [Perry 93], Facile [Thomsen et al. 93] and Erlang [Armstrong et al. 93] emerged; these sacrifice referential transparency by allowing purely functional computations and computations with side effects to be arbitrarily interspersed.

A technique for reconciling referential transparency and side effects does exist, however: monads. In Haskell, all I/O operations are encapsulated in a special monad. Since it is well-known that communicating processes are a generalization of programs performing I/O, in principle, there is no hindrance to use the same technique to enhance Haskell with primitives related to concurrency.

However, a look into the world of concurrent languages reveals a bewildering multitude of communication paradigms and primitives, and even with respect to functional

concurrent languages, the situation is not much clearer. At first glance it seems that concurrency is still very much a moving target, unsuitable to be introduced to Haskell, which is intended to reflect the consensus of the lazy functional programming community. How could a consensus ever be reached regarding the concurrency primitives to be provided?

This paper makes the point that a set of concurrency primitives might be agreed upon if their number and complexity is designed to be *minimal*. Striving for a minimal concurrent extension of Haskell, we present a calculus of only four concurrency primitives that is sufficient to serve as a base for implementing other, arbitrarily more sophisticated, communication paradigms within Haskell, i.e., in a purely equational way. The main part of the paper is devoted to supporting the claim that, using monads, this can be done naturally and systematically. Complete definitions of an Erlang-style actor paradigm and a paradigm in the style of Concurrent ML are given; moreover, an implementation of an Ada-style paradigm is sketched.

We believe that minimizing the number and complexity of the concurrency primitives while maximizing the part of the concurrent program that is defined purely equationally has beneficial effects in the following areas:

---

\*The author's research is supported by a PhD scholarship from the German Research Council (DFG) under grant Ho 1257/1-2.

*Verification:* The fewer and the simpler the primitives, the more of a concurrent program's code is written within the functional language. This means that a maximum of the formal reasoning about its properties can be conducted within the well-understood theory of functional programming, i.e., purely equationally. Only a minimum of the formal reasoning must be conducted within the inherently more complex theory of the underlying concurrency calculus.

*Semantics:* The minimalist approach presented may help to remedy a recurrent problem in the design of concurrent languages, namely, how to keep the language small, coherent, and well-defined, yet to provide the programmer with all the communication constructs required. In reality, concurrent languages seem to fall into two categories: either they rest on secure theoretical foundations but have limited applicability in practice, or they provide all the mechanisms found to be useful in concurrent programming but are so complex as to be difficult to treat formally. An example of the former case is Occam [Inmos Ltd. 84], which is based on the process calculus CSP [Hoare 85], from which it inherits a large body of theoretical properties. Examples of the latter case are the languages SR [Andrews et al. 88] and Ada [Ichbiah 83]. Obviously, the more useful constructs a language contains, the harder it becomes to give a formal semantics to it. Our approach might be a useful compromise. Although giving a formal semantics to the four primitives presented here is beyond the scope of this paper, it should not be impossible. That done, any other programming paradigm defined equationally on top of these primitives would have a well-founded semantics.

*Programming style:* Using monads to structure sequential functional programs amounts to splitting the programming process in two phases that can be characterized as follows: first a customized, application-specific programming language (the monad) is built which takes care of the bookkeeping; then, the interesting part of the program is written in the clearest possible way. We believe that this approach is very promising in concurrent programming, too. In many concurrent programs, a substantial part of the application's code is used for mapping the application's communication structure to the language's primitives.<sup>1</sup> This code can be wrapped up by first programming a monad encapsulating a communication paradigm adequate for the application, then solving the problem.

*Prototyping:* Experiments indicate that functional languages have several advantages over imperative languages for the purpose of prototyping sequential programs [Carlson et al. 93]; it is not clear, though, whether these advantages carry over to the prototyping of concurrent programs. Prototyping of concurrent programs seems to require that the communication paradigm used by the prototype and that provided by the

implementation language are similar; it makes little sense, for instance, to prototype an Ada application in a language based on message passing. The two-phase technique mentioned above may be particularly useful in prototyping: First the concurrency paradigm of the implementation language is modelled in the functional language, then a prototype is built using this paradigm.

The paper is organized as follows: The next section addresses some related work. Section 3 explains the notation used. In Section 4, the four primitives are presented. Building them into Haskell in the style of the I/O monad enables concurrent computations to be expressed in a referentially transparent way. An operational semantics of the primitives is given and illustrated by a sample application. Section 5 shows how operations may be added to an existing communication paradigm: The built-in paradigm is enhanced with additional operations for remote function call. Section 6 through 8 demonstrate how one communication paradigm may be built on top of another; monads encapsulating communication paradigms resembling those paradigms which form the basis of the concurrent languages Erlang, Concurrent ML, and Ada are presented. Section 9 discusses the lessons learned.

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

## 2. Related Work

In [Jones, Hudak 93], the potential of using monads to express concurrent computations is explored. To this end, three operations corresponding to our *newChan*, *send* and *receive* primitives are added to the IO monad. The main difference to our work seems to be that Jones and Hudak aim at avoiding nondeterministic semantics. Therefore, they do not make *fork* a primitive which introduces nondeterminism into the language, but rather define it as an ordinary function within the language which serves as an annotation telling the compiler to try to execute two expressions in parallel. While Hudak and Jones highlight the advantages of their approach by comparing them to lazy stream-processing, our work is more in the tradition of [Karlsson 81] and [Perry 90], who use continuations for introducing nondeterminism and concurrency into pure functional languages without sacrificing referential transparency. Here, the main objective is not to speed up functional computations by parallel execution in a context where nondeterminism is considered a necessary evil, but rather to enable concurrent applications which are inherently nondeterministic, for instance operating systems and graphical user interfaces, to be expressed in a functional language. We focus on a question, though, that as yet seems to have received little attention in the literature, namely, how the choice of the concurrency primitives influences the software engineering of concurrent programs in functional languages.

---

<sup>1</sup> In concurrent programming, a saying - originally coined to promote monads in another context - seems to fit, namely, that "programmers often find themselves peering through the underbrush at the interesting code somewhere within." [Hall et al. 92].

### 3. Notation

The notation used in this paper is essentially the functional programming language HASKELL [Hudak et al. 92], however, with the following extensions:

We assume 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]. The syntax for monads 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 transforming a *do* expression into one using ``bind`` are given in Fig. 1. In case more than one rule matches, the first one applies.

<code>do { Pat ← Exp; Rest }</code>	$\Rightarrow$	<code>Exp `bind` \Pat → do { Rest }</code>
<code>do { Exp; Rest }</code>	$\Rightarrow$	<code>Exp `bind` \_ → do { Rest }</code>
<code>do { let { .. }; Rest }</code>	$\Rightarrow$	<code>let { .. } in do { Rest }</code>
<code>do { [Exp] }</code>	$\Rightarrow$	<code>result Exp</code>
<code>do { Exp }</code>	$\Rightarrow$	<code>Exp</code>

Fig 1: Special syntax for monads

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

In this syntax, the following expression using primitives from the Haskell 1.3 proposal for monadic I/O

```
getChar `bind` \c →
(case c of 'y' → putChar 'Y'
          'n' → putChar 'N') `bind` \_ →
result True
```

becomes

```
do c ← getChar
   case c of 'y' → putChar 'Y'
            'n' → putChar 'N'
   [True]
```

Moreover, we take the licence to declare type synonyms as instances of constructor classes. In Gofer, this is only possible for type constructors of algebraic data types, which means that every type synonym that is to be used as a monad must be enclosed in an algebraic data type of its own, which leads to a slight cluttering of the code.

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].

### 4. The Concurrency Primitives

#### 4.1. Interface

The concurrency primitives that are added to Haskell are represented by functions on a monad *Process*. Their signatures are given in Fig. 2.

<code>send</code>	$::$	<code>Chan a → a → Process ()</code>
<code>receive</code>	$::$	<code>Chan a → Process a</code>
<code>fork</code>	$::$	<code>Process () → Process ()</code>
<code>newChan</code>	$::$	<code>Process (Chan a)</code>

Fig. 2: The concurrency primitives

A term of type *Process a* is called a *process term*. It encapsulates a concurrent computation yielding a value of type *a*. In analogy to the *IO* monad, we assume that the compiler will know that process terms must be evaluated according to the operational semantics given in Section 3.3. At any time, an arbitrary number of process terms will be evaluated concurrently. Each of these terms is called a *process*.

In general, monads representing a given communication paradigm are abstract data types with a given set of interface functions (besides *bind* and *result*). In the sequel, we will call these interface functions the paradigm's *instructions*. Other functions on the monad which are defined in terms of the instructions are called *operations*.

Processes communicate by point-to-point, synchronous message-passing on typed, first-class channels.

The operations a process can perform are: send a message on a given channel (*send*) of matching type, receive a message on a channel (*receive*), start another process (*fork*), or create a new channel (*newChan*). A process wanting to send a message on a given channel is halted until a process wanting to receive a message on the same channel is found, and vice versa, such that the sender and the receiver of a message are forced to synchronize.

In case that at one time there are many processes wanting to execute a *send* or a *receive* operation on the same channel, they are paired in a nondeterministic manner that is at the implementation's discretion. It is guaranteed, however, that every message sent is received exactly once by one process. Note that the data type *Chan a* is an abstract data type which has a representation that depends on the language implementation. Besides equality, there is only one other function named *toInt* defined on objects of *Chan a*. *toInt* returns different integer values for different channels.

In the sequel, it will be convenient to overload the primitives' names. Thus, we will assume a type class *ProcessMonad* containing all monads on which the primitives are defined.

Moreover, we will assume that the compiler knows that programs cannot only be expressions of type  $IO ()$ , like in Haskell, but also of type  $ProcessMonad p \Rightarrow p a$ . Since the compiler knows an operational semantics only for type  $Process$ , other instances of class  $ProcessMonad$  must supply a function  $toProcess$ , which the compiler uses to transform the top-level expression into a process.

```
class Monad p \Rightarrow ProcessMonad p where
  send    :: Chan a \to a \to p ()
  receive :: Chan a \to p a
  newChan :: p (Chan a)
  fork    :: Process () \to p ()

  toProcess :: p a \to Process a
```

Of course,  $Process$  is a member of this class.

```
instance ProcessMonad Process where
  send    = primSend
  receive = primReceive
  newChan = primNewChan
  fork    = primFork

  toProcess = id
```

## 4.2. Implementation

Since the outcome of a concurrent computation is nondeterministic, the monad  $Process$  cannot be implemented within Haskell. We describe the operational semantics of the primitives by a transition relation on sets of process terms.

In Fig. 3 where the primitives' operational semantics is defined,  $exp [v := exp']$  denotes the expression obtained by substituting every free occurrence of variable  $v$  in expression  $exp$  by  $exp'$ . The operator  $\otimes$  denotes the union of two sets with empty intersection.

<b>Initialization:</b>	
$do\ p$	$\Rightarrow \{do\ p\}$ where $p :: Process\ a, a \neq ()$
<b>Transition:</b>	
$ps \otimes \{do\ send\ m\ ch; p\}$	$\otimes \{do\ m' \leftarrow receive\ ch; p'\} \Rightarrow ps \cup \{p, p'[m' := m]\}$
$ps \otimes \{do\ [()]\}$	$\Rightarrow ps$
$ps \otimes \{do\ fork\ p'; p\}$	$\Rightarrow ps \cup \{p', p\}$
$ps \otimes \{do\ ch \leftarrow newChan; p\}$	$\Rightarrow ps \cup \{p[ch := ch']\}$ where $ch'$ fresh
<b>Successful termination:</b>	
$ps \otimes \{do\ [x]\}$	$\Rightarrow x$ where $x :: a, a \neq ()$

**Fig. 3:** The concurrency primitives' operational semantics

To start the evaluation of a process term  $p$ , an initial set having  $p$  as its only element is created. Repeatedly, from the transition rules given in Fig. 3, one that matches the current set is selected nondeterministically and applied to the current

set, yielding a new set. This procedure is repeated until either the initial process has been reduced to normal form  $[x]$  (which stands for *result*  $x$  in our syntax!), or there is no matching rule. In the former case, the computation's result is  $x$ , in the latter case, the computation is deadlocked. Note that the initial process has type  $Process\ a$  where  $a \neq ()$  while other processes created in the course of the computation have type  $Process\ ()$ . Moreover, note that there is no explicit termination command: processes terminate implicitly when they return a value.

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. Two techniques for constructing new communication mechanisms are studied in this paper: on the one hand, an existing communication paradigm can be extended with additional operations; on the other hand, a completely new communication paradigm may be constructed on top of an existing one. In the remainder of this section and in the next one, additional operations on the  $Process$  paradigm will be developed; the remaining sections are devoted to the implementation of new paradigms.

## 4.3. Application

To illustrate the use of the concurrency primitives, we develop a generic concurrent divide-and-conquer operation  $divAndConq$ . At the core of every divide-and-conquer algorithm there are four domain-specific functions which govern the algorithm's behaviour:  $isSolvable$  is used to determine whether a problem is solvable, in that case it can be solved by  $solve$ ; otherwise it must be divided into two simpler subproblems (using  $divide$ ), whose solutions are combined using  $compose$ . These functions may conveniently be abstracted from using type classes.

```
class Problem p where isSolvable :: p \to Bool
                      divide    :: p \to (p, p)

class Solution s where compose :: s \to s \to s

class Solvable p s where solve  :: p \to s
```

The operation  $divAndConq$  takes a problem as its parameter, creates a subprocess to calculate the solution, and immediately returns a newly-created channel  $sChan$ , on which the subprocess will eventually make the solution available. In case the problem is trivial, the subprocess sends the solution on  $sChan$  and terminates. Otherwise, it divides the problem into two subproblems  $p1$  and  $p2$ , solves them using  $divAndConq$ , receives the subsolutions on channels  $sChan1$  and  $sChan2$ , composes them, and terminates after sending the final solution on  $sChan$ .

$divAndConq :: (Problem\ p, Solution\ s, Solvable\ p\ s) \Rightarrow$   
 $p \rightarrow Process\ (Chan\ s)$

```
divAndConq p =
  do sChan ← newChan
  fork (
    do if isSolvable p then
      do send sChan (solve p)
    else
      do let (p1,p2) = divide p
          sChan1 ← divAndConq p1
          sChan2 ← divAndConq p2
          s1 ← receive sChan1
          s2 ← receive sChan2
          send sChan (compose s1 s2))
  [sChan]
```

Taking lists of integers to be the problem and solution domain, the following instantiation of the classes *Problem*, *Solution* and *Solvable* turns *divAndConq* into the concurrent version of a popular sorting algorithm.

```
instance Problem [Int] where
  isSolvable    = (<= 1) . length
  divide (x:xs) = if null as then ([x], bs)
                  else (as, x:bs)
  where
    as = [ x' | x' ← xs, x' <= x ]
    bs = [ x' | x' ← xs, x' > x ]

instance Solution [Int] where compose = (++)
instance Solvable [Int] [Int] where solve = id

quickSort :: [Int] → Process [Int]
quickSort p =
  do sChan ← divAndConq p
  receive sChan
```

Note that *length*, *null*, *id*, and *(++)* are defined in the standard prelude. *length* returns the length of a list, *null* tests whether a list is empty, *id* is the identity function, and *(++)* concatenates two lists.

## 5. Adding Remote Function Call

Given the purely functional definition of an abstract data type (abbrev. ADT), we extend the *Process* monad with operations for defining multiple server processes, each with a unique identity, offering the ADT's operations as remote functions to arbitrary client processes.

### 5.1. Interface

The interface of the remote function call mechanism consists of two operations *newServer* and *(?)*.

```
newServer :: a → Process (ServerId a)
(?) :: ServerId a → (a → (b,a)) → Process b
```

The function *newServer* 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 applied to the object managed by the server process, which updates its state accordingly, and sends an object of type *b* back to the client.

### 5.2. Application

Consider the ADT *Dictionary* which offers dictionary services.

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

Its interface consists of one generator function *createDictionary* and three operations *add*, *delete*, and *lookUp*. (Their definitions are omitted here.) The following process *rfcClient* illustrates the creation and use of a remote function call server for objects of type *Dictionary*.

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

Note that the derived process functions coexist with the interface functions of *Process*, i.e., one process may use both, for instance, *(?)* and *send*.

### 5.3. Implementation

Conceptually, a server for an object of a given type *a* is a process having the object as its state. It receives functions of type *a → (b, a)*, applies the function to its state, returns the first element of the resulting pair to the caller and update its state with the second element. The problem is how to represent a reference to the server. A general technique is to identify a process by the channel it listens to. Thus, the straightforward solution is to implement *ServerId a* as follows:

```
type ServerId a = Chan (a → (b, a), Chan b) -- wrong
```

This would mean that a server understands messages consisting of a state transformer function to be executed and a channel on which to return the result. However, this does not work without existential types, because, for different requests, type *b* is expected to vary. How can type *b* be removed from the definition of type *ServerId a*? The solution is to have the requester transmit a message which combines the application of the state transformer and the command sending back the result into one piece of code:

```
type ServerId a = Chan (a → Process a) -- ok
```

To issue a request using *(?)*, a process sends such a piece of code to the server which takes the server's state *a* as an

argument, applies the state transformer to it, yielding an object  $(b, a')$ , makes  $b$  available on a channel that the requester listens to, and returns  $a'$ .

```
(?) :: ServerId a → (a → (b,a)) → Process b
requestChan ? f =
  do replyChan ← newChan
     send requestChan (\a → do let (b, a') = f a
                               send replyChan b
                               [a])
     receive replyChan
```

The server repeatedly receives such a piece of code, provides it with its state and gets back its new state when executing it.

```
server :: ServerId a → a → Process ()
server requestChan a =
  do request ← receive requestChan
     a' ← request a
     server requestChan a'
```

`newServer`  $a$  is implemented by forking a server process for  $a$ , supplying it with a new channel on which to listen, and returning the appropriately typed channel.

```
newServer :: a → Process (ServerId a)
newServer a =
  do requestChan ← newChan
     fork (server requestChan a)
     [requestChan]
```

#### 5.4. Remarks

Note that the primitives `newVar`, `readVar`, and `writeVar` used by Launchbury and Peyton Jones to incorporate mutable state variables into Haskell [Launchbury, Peyton Jones 94] can be implemented trivially in terms of `newServer` and `(?)`.

## 6. Monad transformers

In the following chapters, we will construct new monads, capturing more sophisticated communication paradigms on top of the `Process` monad. This will be done by adding new features to an existing monad. The following type classes `StateMonad`, `ReadMonad` and `FailMonad` identify monads on which operations for manipulating mutable state, for accessing read-only state, and for performing failure exits are defined.

```
class Monad m ⇒ StateMonad st m where
  get :: m st
  set :: st → m ()

class Monad m ⇒ FailMonad m where
  fail :: m a

class Monad m ⇒ ReadMonad r m where
  read :: m r
```

Features may be added to existing monads by means of so-called monad transformers.

```
class MonadTransformer t where
  lift :: Monad m ⇒ m a → t m a
```

The monad transformer `S st` enhances an existing monad  $m$  with mutable state of type  $st$ :

```
type S st m a = st → m (a, st)

instance Monad m ⇒ Monad (S st m) where
  result a = \st → [(a, st)]
  bind ma f = \st → do (a, st') ← ma st; f a st'

instance StateMonad st (S st m) where
  get = \st → [(st, st)]
  set st' = \_ → [((, st'))]

instance MonadTransformer (S st) where
  lift ma = \st → do a ← ma; [(a, st)]
```

The monad transformer `R r` makes a read-only value of type  $r$  accessible to an existing monad  $m$ .

```
type R r m a = r → m a

instance Monad m ⇒ Monad (R r m) where
  result a = \_ → [a]
  ma `bind` f = \r → do a ← ma r; f a r

instance ReadMonad r (R r m) where
  read = \r → [r]

instance MonadTransformer (R r) where
  lift ma = \_ → do a ← ma; [a]
```

The monad transformer `F` enhances an existing monad  $m$  with an operation for performing a failure exit.

```
type F a = m (Maybe a)

data Maybe a = Yes a
             | No

instance Monad m ⇒ Monad (F m) where
  result a = [Yes a]
  ma `bind` f = do maybe ← ma
                 case maybe of
                   No → [No]
                   Yes a → f a

instance FailMonad (F m) where
  fail = [No]

instance MonadTransformer F where
  lift ma = do a ← ma; [Yes a]
```

For a more detailed presentation of the techniques for combining monads, consult [King, Wadler 92], [Jones, Duponcheel 93], and [Jones 95].

## 7. Actors

In the previous section, one communication paradigm was enhanced with additional operations. This section illustrates how a completely new communication paradigm, an actor paradigm resembling the one on found in Erlang [Armstrong et al. 93], may be built on top of an existing one, namely, the built-in calculus.

## 7.1. Interface

The actor monad *ACT* implements the paradigm of point-to-point, unidirectional, asynchronous, buffered message passing with explicit, guarded message receipt using an asymmetric naming scheme. That is, sending a message is non-blocking and requires the sender to know the name of the receiver. An actor 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 an actor are buffered in an unbounded message queue. The order in which two messages were sent is not necessarily that in which they arrive. Each actor in the system is uniquely identified by a process identifier (abbrev. *PID*).

An expression of type *ACT m a* is called an *actor term* and represents a computation which understands messages of type *m* and returns a result of type *a*. (The distinction between *actors* and actor terms is the same as the distinction between processes and process terms.) These are the operations an actor can perform:

```

receiveACT  :: (m → Bool) → ACT m m
sendACT    :: Id m' → m' → ACT m ()
selfACT    :: ACT m (Id m)
forkACT    :: ACT m' () → ACT m (Id m')
```

*sendACT* takes the receiving actor's *PID* and the data item to be transmitted as its parameters. *receiveACT* takes a predicate on messages, the guard, as its parameter; it blocks until a message *m* arrives for which the guard evaluates to *True*, then it returns *m*. *selfACT* take no parameters and immediately returns the current actor's *PID*. *forkACT* takes the actor term to be evaluated concurrently as its parameter and returns the child actor's *PID*. Note that *PIDs*, like channels, are typed, such that no actor can receive a message that it does not understand.

## 7.2. Application

This example actor returns the value *100*. It illustrates how the use of guards enables an actor to process messages in an order that is independent of their order of arrival. Moreover, it illustrates that the operation *sendACT* is non-blocking.

```

mainACT :: ACT Int Int
mainACT =
  do self ← selfACT
     child ← forkACT (do sendACT self 50
                       sendACT self 150)
     sendACT child "hallo"
     a ← receiveACT (> 100)
     b ← receiveACT (< 100)
     [a - b]
```

Note that the implementation of the actor calculus is completely hidden. In particular, an actor cannot execute operations defined on *Process*.

## 7.3. Implementation

It is well-known that asynchronous communication can be implemented in terms of synchronous communication by use of buffers. The idea is to implement an actor by a process with additional read-only state, namely, a channel on which to listen for messages, and additional mutable state, namely, a message buffer.

```

type Actor m = R (Chan m) (S [m] Process)

instance StateMonad [m] (Actor m) where
  set = lift . set
  get = lift get

instance ProcessMonad (Actor m) where
  send chan m = lift (lift (send chan m))
  receive chan = lift (lift (receive chan))
  fork p      = lift (lift (fork p))
  newChan     = lift (lift newChan)

toProcess    = do chan ← newChan
               (a, _) ← p chan []
               [a]
```

The initial actor is started with an empty message queue.

An actor is identified by its *ID*, which is implemented by the channel the actor listens to.

```
type Id m = Chan m
```

To receive a message, an actor checks whether its message buffer contains a message for which the guard evaluates to true, in which case this message is removed from the buffer and returned immediately. Otherwise, the actor repeatedly executes *receive* and places the arriving messages in the buffer until a message arrives for which the guard evaluates to true.

```

receiveACT :: (m → Bool) → ACT m m
receiveACT guard =
  do chan ← read
     xs ← get
     let (xs', xs'') = span (not . guard) xs
         if null xs'' then
           do let loop =
                do xs ← get
                   x ← receive chan
                   if guard x then
                     do [x]
                   else
                     do set (xs ++ [x])
                        loop
               loop
         else
           do set (xs' ++ tail xs'')
              [head xs'']
```

Note that *span* is defined in the standard prelude. It takes a predicate and a list as its parameters and splits the list into a left and a right part. The left part is the largest initial part of

the list such that the predicate is true for all its elements; the right part is the rest of the list.

To send a message to another actor, the sender forks a process that makes the message available on the channel that the receiver is listening to. This way, the sending actor does not block until the message has been processed by the receiver.

```
sendACT :: Id m' → m' → ACT m ()
sendACT other m =
  do fork (send other m)
```

Using *selfACT*, an actor gets to know its own PID.

```
selfACT :: ACT m (Id m)
selfACT =
  do chan ← read
     [chan]
```

When a new actor is forked, it is supplied with a new channel to listen to and an empty message buffer. Its state after termination is discarded.

```
forkACT :: ACT m' () → ACT m (Id m')
forkACT p =
  do chan ← newChan
     fork (do p chan []; [()])
     [chan]
```

## 8. Concurrent ML

In this section, the communication paradigm used in Concurrent ML is implemented on top of the built-in paradigm. Essentially, this means adding an operator for external choice, as found in, e.g., Occam and Facile. However, these languages require the guards of the external choice to be syntactically distinguished from ordinary send or receive commands. This requirement severely reduces the modularity of the resulting programs; this topic is discussed at length in [Reppy 88]. In Concurrent ML, and in our calculus, this restriction does not apply.

### 8.1. Interface

The *CML* paradigm has four instructions which have the same semantics as those of the built-in paradigm:

```
sendCML    :: ChanCML a → a → CML ()
receiveCML :: ChanCML a → CML a
forkCML    :: CML () → CML ()
newChanCML :: CML (ChanCML a)
```

However, there is one additional instruction:

```
chooseCML :: [CML a] → CML a
```

While each ordinary process is only ready to execute one instruction at a time, a *CML* process having the form *chooseCML*  $[p_1, \dots, p_n]$  may be ready to execute more than one instruction at a time, namely, all the first instructions of the  $p_1, \dots, p_n$ . The choice which one is actually executed is nondeterministic.

### 8.2. Application

The string returned by the following *CML* process *mainCML* is either "*p1: first p2: first*" or "*p1: second p2: second*".

```
mainCML :: CML String
mainCML =
  do chan1 ← newChanCML
     chan2 ← newChanCML
     chan3 ← newChanCML

  let p1 =
        do alt ← chooseCML [
            do sendCML chan1 1
               ["p1: first "],
            do receiveCML chan2
               ["p1: second "]]
        ]
        sendCML chan3 alt

    p2 =
        do alt ← chooseCML [
            do chooseCML [
                do receiveCML chan1
                   ["p2: first "],
                do chooseCML [
                    do sendCML chan2 2
                       ["p2: second "],
                    do id p1
                       ["Can't happen!"]]]]]
        ]
        sendCML chan3 alt
    chooseCML [forkCML p1]
  forkCML p2
  s1 ← receiveCML chan3
  s2 ← receiveCML chan3
  [s1 ++ s2]
```

The process illustrates two points about the paradigm:

1. The nesting of *chooseCML* instruction does not matter: *p2* could be rewritten using only one *chooseCML* instruction.
2. An element of a *chooseCML* instruction is not restricted to an instruction sequence starting with a *send* or a *receive*, but may be an arbitrary functional expression, like *id p1*, or *forkCML p2*.

The last point is especially important for enabling abstraction and modularity, which was first recognized by Reppy and motivated him to devise "first-class synchronous actions" [Reppy 88] which form the basis of Concurrent ML [Reppy 93]. In Concurrent ML and in the calculus presented here, one can compose server processes  $p_1, \dots, p_n$  to a new server process writing *chooseCML*  $[p_1, \dots, p_n]$ , without knowing on which channels  $p_1, \dots, p_n$  want to communicate, whether they

want to send or to receive, or knowing anything at all about their implementation. In fact, the guards of the external choice operator may be computed dynamically. This an advantage over languages like Occam or Facile, where the guards of the external choice operator must be known statically.

### 8.3. Implementation

The idea of the implementation is to have a global *transaction manager* process which any *CML* process consults before executing an instruction. This is done by sending a *transaction request* to the transaction manager.

```

type TReq = (Tid, Chan (Maybe Tid), TAKind)

data TAKind = Send ChanId (Process ())
            / Receive ChanId
            / Fork (Chan Tid)
            / NewChan

type Tid = Int

data Maybe a = Yes a
            / No

```

A process issuing a transaction request tags it with its *current transaction identifier* (abbrev. TID). The transaction manager can either *commit*, or *abort*, or *suspend* a transaction request. For every TID, it will commit at most one transaction request tagged with this TID and abort all the others. The *chooseCML* instruction is implemented by forking one *CML* process for each of its element terms. While a process's current TID is usually unique, all processes created to evaluate an element term of a *chooseCML* instruction have the same current TID. In order to execute their first instruction, they all issue transaction requests tagged with the same TID. Eventually, only one of them will be committed, which means it can proceed; all others will be aborted, which means they terminate.

The transaction manager issues the TIDs to the processes; whenever one transaction request is committed, the requesting process is handed a fresh TID. The transaction manager keeps track of the set of valid TIDs. For each transaction request that commits, the TID of the request is removed from the set of valid TIDs and the fresh TID which was handed to the requesting process is added to the set. Moreover, the transaction manager stores all suspended transaction requests, i.e., *Send* or *Receive* requests with no matching request

The transaction manager can be in one of three states: its *initial* state, the *commit* state, or the *purge* state. In the initial state, the transaction manager waits for a transaction request. If the request's TID is not valid at all, i.e., if a transaction request with this TID has committed already, the transaction is aborted straightaway and the transaction manager returns to its initial state. In case the TID is valid, the transaction manager must decide whether to commit the transaction or suspend it. If the request is neither for a *Fork* nor a *NewChan* transaction, it is committed immediately; the transaction manager goes into state *commit*. If the request is for a *Send* or

a *Receive* transaction, however, it can only commit together with a matching request. Two transaction match, if one of them is a *Receive* transaction, the other is a *Send* transaction, they both want to communicate on the same channel, and they don't belong to the same process.

```

matches :: TReq → TReq → Bool
matches (tid, _, Send ch) (tid', _, Receive ch') =
  (ch == ch') && (tid /= tid')
matches (tid, _, Receive ch) (tid', _, Send ch') =
  (ch == ch') && (tid /= tid')
matches _ _ =
  False

```

This is the transaction manager's code:

```

taManager :: Chan TReq → [TReq]
           → [Tid] → Tid → Process ()

taManager inChan suspReqs validTids nextTid =
  do (req @ (tid, replyChan, kind)) ← receive inChan
     if not (tid `elem` validTids) then
       do abort req
          taManager inChan suspReqs validTids nextTid
     else
       do case kind of
          Fork _ →
            commit [req] suspReqs
          NewChan →
            commit [req] suspReqs
          _ →
            case span (not . matches req) suspReqs of
              (_, []) →
                taManager inChan (req:suspReqs)
                          validTids nextTid
              (reqs1, req':reqs2) →
                commit (sort [req, req'])(reqs1 ++ reqs2)

```

where

```

commit [(tid, replyChan, Fork replyChan')] suspReqs' =
  do send replyChan (Yes nextTid)
     send replyChan' (nextTid + 1)
     purge [tid] suspReqs' 2
commit [(tid, replyChan, NewChan)] suspReqs' =
  do send replyChan (Yes nextTid)
     purge [tid] suspReqs' 1
commit [(tid, replyChan, Receive _),
        (tid', replyChan', Send _ sendCom)] suspReqs' =
  do send replyChan (Yes nextTid)
     send replyChan' (Yes (nextTid + 1))
     purge [tid, tid'] suspReqs' 2

purge commitIds suspReqs' idsUsed =
  do let stillOk (tid, _, _) = not (tid `elem` commitIds)
     for (filter (not . stillOk) suspReqs') abort
     taManager inChan
       (filter stillOk suspReqs')
       ((validTids \\\ commitIds) ++
        [nextTid..nextTid+idsUsed-1])
       (nextTid + idsUsed)

```

Note that *for* is a generic function on monads which is defined as follows:

```

for :: [a] → (a → M b) → M [b]
for [] f = [[]]
for (a:as) f = do b ← f a; bs ← for as f; [b:bs]

```

If there is already a matching request in the transaction manager's list of suspended requests then both the incoming and the matching requests are committed simultaneously. Otherwise, the incoming request is suspended.

Committing one or two transaction requests in state *commit* always involves sending a fresh TID to the requester. In the case of a *Fork* request, an additional fresh PID for initializing the child process must be sent to the requester. After that, those suspended transaction requests that have the same TID as the one just committed are aborted and removed from the store; this is done in state *purge*.

```

abort :: TReq → Process ()
abort (_, replyChan, _) = send replyChan No

```

A *CML* process term is implemented by an ordinary process term with extra mutable state, namely, its current TID, and extra read-only state, namely, the channel on which the transaction manager receives requests. Moreover, a *CML* process needs an error exit to allow for the possibility of its current transaction being aborted.

```

type CML = R TManagerId (S Tid (F Process))

instance StateMonad Tid CML where
  get = lift get
  set = lift . set

instance FailMonad CML where
  fail = lift (lift fail)

instance ProcessMonad CML where
  send chan m = lift (lift (lift (send chan m)))
  receive chan = lift (lift (lift (receive chan)))
  fork p = lift (lift (lift (fork p)))
  newChan = lift (lift (lift newChan))

toProcess p = do taChan ← newChan
               fork (taManager taChan [] [1] 2)
               Yes (a, _) p taChan 1
               [a]

```

This is how the initial *CML* process is started: First, the transaction manager is started in a state where 1 is the only valid TID and 2 is the next TID to be issued. Then, the initial *CML* process is started with an initial TID of 1.

A process wanting to conduct a transaction of kind *kind* executes instruction *tryTA kind*. A message composed of the process's current TID, a reply channel, and *kind* is then sent to the transaction manager. In case the reply from the transaction manager is negative, the process fails itself. If the

reply is positive, it contains a the process's new TID which is to be used for executing the next instruction.

```

tryTA :: TKind → CML ()
tryTA kind =
  do id ← get
     tm ← read
     replyChan ← newChan
     send tm (id, replyChan, kind)
     reply ← receive replyChan
     case reply of
       Yes id' →
         do set id'
            No →
              do fail

```

The transaction request for *sendCML* and *receiveCML* contain an integer value encoding the channel on which a value is to be communicated.

```

sendCML :: ChanCML a → a → CML ()
sendCML chan a =
  do tryTA (Send (toInt chan))
     send chan a

receiveCML :: ChanCML a → CML a
receiveCML chan =
  do tryTA (Receive (toInt chan))
     receive chan

```

To fork a new *CML* process, it is initialized with a fresh TID and the channel that the transaction manager is listening to. The resulting state is discarded.

```

forkCML :: CML () → CML ()
forkCML code =
  do replyChan ← newChan
     tryTA (Fork_ replyChan)
     id ← receive replyChan
     tm ← read
     fork (do Yes ((), _) ← code tm id
           [()])

```

A *CML* channel is implemented by an ordinary channel.

```

type ChanCML = Chan

```

Channel creation in the *CML* paradigm is the same as in the built-in paradigm

```

newChanCML :: CML (ChanCML a)
newChanCML =
  do tryTA NewChan
     newChan

```

For each of its element terms, *chooseCML* forks one process, which executes the element process and sends the result and he current TID at the time of termination termination on a common channel *aChan*. Since the first transaction of all but

one process will be aborted, only one value will ever become available on *aChan*.

```

chooseCML :: [CML a] → CML a
chooseCML pas =
  do id ← get
     tm ← read
     aChan ← newChan
     for pas (\pa →
       do fork (
         do res ← pa tm id
            case res of
              Yes (a, id') → send aChan (a, id')
              No           → [()])
         (a, id') ← receive aChan
         set id'
         [a]

```

## 9. Ada

The last paradigm presented in this paper is an attempt to capture some of the essentials of concurrent tasks in Ada. Tasks communicate by accepting and requesting service rendezvous from each other on the basis of an asymmetric naming scheme where the requesting task must specify the desired server, while a task wanting to accept a service request cannot decide which task it is ready to serve.

### 9.1. Interface

These are the operations that a process may perform:

```

newService  :: ADA (Service x y)
selfADA     :: ADA TaskId
forkADA     :: ADA () → ADA TaskId
requestADA  :: TaskId → Service x y → x → ADA y
acceptADA   :: [Alt] → ADA ()

```

*newService* creates a new service reference, i.e., a typed object which is used to tag and match service offers and service requests. *selfADA* and *forkADA* have functionalities analogous to *selfACT* and *forkACT*. *requestADA* takes a server's ID, an operation identifier and the operation's argument as its parameters. *requestADA* blocks until the chosen server has accepted and executed the operation, then it returns the operation's result. *acceptADA* takes a list of alternatives as its argument. Each alternative associates one operation identifier with a piece of code handling this operation. An alternative is constructed using operator ( $\gg$ ).

$(\gg) :: Service\ x\ y \rightarrow (x \rightarrow ADA\ y) \rightarrow Alt$

### 9.2. Application

The program *mainADA* uses a server task *calculator* whose exceedingly weird behaviour can only be justified with its intended use of illustrating the communication paradigm.

```

mainADA :: ADA Int
mainADA =
  do mult ← newService
     square ← newService

  let calculator :: ADA ()
      calculator =
        do acceptADA [
          mult >> \ (m, n) →
            do [m * n]
          ]
          acceptADA [
          mult >> \pair →
            do helper ← forkADA calculator
               requestADA helper mult pair,
            square >> \n →
              do acceptADA [
                square >> \n' →
                  do [n'*n']
                ]
              [n*n]
          ]
        calculator

  server ← forkADA calculator
  a ← requestADA server mult (4,5)
  forkADA (requestADA server square 0)
  requestADA server square a

```

Initially, a *calculator* task offers only one kind of service to its surroundings, namely, the operation *mult*. Having successfully executed it, it offers the operations *mult* and *square*. Should the surroundings choose the operation *mult* to be executed, the calculator starts another calculator and lets it perform the actual multiplication. Should a client task choose the *square* service, it will be blocked until a second client asks for the square service, too; then, both clients will be served and the server will return into its initial state.

Note that this paradigm does not allow for postprocessing to be done, i.e., computations being performed which depend on the service accepted, but taking place after the service's result has been transmitted to the caller. This can be easily remedied by changing the signatures of *acceptADA* and ( $\gg$ ) to become

```

acceptADA :: [Alt a] → ADA a
(\gg)     :: Service x y → (x → ADA (y, ADA a)) → Alt a

```

### 9.3. Implementation

Since the implementation of the ADA paradigm introduces new techniques, it is omitted. Each *Service* object is implemented by a process which matches service requests and service offers in a manner that is similar to, albeit simpler than, that of the transaction manager. The difference is that the decision which transactions to commit and which to abort

need not be made by one global authority but can be made by the task executing *acceptADA*, since a *requestADA* statement is not allowed to contain alternatives. The code required for implementing the *ADA* paradigm is slightly more than half the size of the *CML* paradigm's code.

## 10. Conclusions

The aim of the work that has been presented in this paper is to design a *minimal concurrent extension* for Haskell and to assess its usefulness. To this end, four very simple concurrency primitives were built into Haskell using the technique known from monadic I/O. Apart from assuming language support for monads, as found in Gofer, no changes were made to Haskell; in particular, no new language features related to concurrency were introduced.

To us, the preliminary conclusions of our work are the following:

- Neither laziness, nor static typing, nor referential transparency need be sacrificed on the altar of concurrency.
- In terms of expressive power and readability, programs written using current Haskell technology can compete with programs written in concurrent functional languages like Facile, Erlang and Concurrent ML.

Moreover, it seems that the two-phase approach to programming encouraged by monads has some particularly worthwhile applications in concurrent programming.

## 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
- [**Armstrong et al. 93**] J. Armstrong et al.: *Concurrent Programming in Erlang*, Prentice-Hall, 1993
- [**Carlson et al. 93**] W.E. Carlson, P. Hudak, M.P. Jones: *An Experiment Using Haskell to Prototype "Geometric Region Servers" for Navy Command and Control*, Research Report 1031, Dept. of Computer Science, Yale University, 1993
- [**Hall et al. 92**] C. Hall et al.: *The Glasgow Haskell Compiler: A Retrospective*, 1992 Glasgow Workshop on Functional Programming, Ayr, 1992

[**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

[**Ichbiah 83**] J. Ichbiah (ed.): *Ada Programming Language*, ANSI-MIL-STD-1815A, Ada Joint Program Office, Department of Defense, Washington DC, 1983

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

[**Jones, Duponcheel 93**] M. Jones, L. Duponcheel: *Composing Monads*, Research Report YALEU/DCS/RR-1004, Yale University, 1993

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

[**Jones 95**] M. Jones: *Functional Programming with Overloading and Higher-Order Polymorphism*, First International Spring School on Advanced Functional Programming Techniques, LNCS 925, Springer Verlag, 1995

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

[**King, Wadler 92**] D.J. King, P. Wadler: *Combining Monads*, 1992 Glasgow Workshop on Functional Programming, Ayr, 1992

[**Reppy 88**] J.H. Reppy: *Synchronous Operations as First-Class Values*, ACM SIGPLAN Conference on Programming Language Design and Implementation, 1988

[**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

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

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