Explicit Message Passing for Concurrent Clean

Pascal R. Serrarens

Computer Science Institute
University of Nijmegen, The Netherlands
pascalrs@cs.kun.nl

Abstract. In this paper we look at Concurrent Clean and concurrency and notice that the language has some shortcomings with respect to communication. It does not provide non-determinism, efficient multicasting and data-driven communication. A message passing extension for Concurrent Clean is proposed which provides efficient many-to-many communication. In contrast to other solutions, we chose to have an asynchronous system where sending and receiving do not have to occur simultaneously. We discuss design decisions with respect to concurrent evaluation and unique messages. Furthermore, we show some implementation aspects of this message passing system.

1 Introduction

An important topic in computer science is concurrency. It is getting more attention nowadays with the introduction of shared memory machines, like multi-Pentium computers. For these systems, concurrency is often provided through a library which enables the programmer to create processes or threads and have communication between them. In that case, concurrency is outside the language, but we also have a number of languages where concurrency is integrated. An example is Pict [Tur95]

Functional languages designers have a great interest in concurrency, especially in relation to graphical user interfaces. Because of the absence of side-effects, functional languages are ideal for parallelism and concurrency. The problem is that concurrency often introduces non-determinism into the system, which complicates reasoning. Several attempts have been made to introduce concurrency without non-determinism [HC93] [JH93], but they have not been a real success. It seems that non-determinism is really needed to describe systems like multi-user systems.

Concurrent Clean also provides threads and communication, through process annotations and lazy graph copying respectively. However, experience showed that this is not powerful enough to describe complex concurrent systems in a clear way. Additional power is needed though a flexible message passing system. This paper presents such a system.

In Section 2 the current concurrency constructs of Concurrent Clean are discussed. Section 3 then summarises the shortcomings of these constructs. After looking at some other solutions in Section 4, we look in Section 5 how message passing can be added to Concurrent Clean. Section 6 discusses some implementation issues. Section 7 discusses current work on message passing, while Section 8 concludes.

2 Clean and Concurrency

2.1 Process Annotations

Concurrent Clean [NSvEP91] provides a number of process annotations, which are used to create threads in a Clean program. These threads execute concurrently with all other threads: either parallel, on a different processor, or in an interleaved manner on the same processor.

There are three kinds of annotations:

- `{| I |}`: the expression is reduced interleaved on the same processor.
- `{| P |}`: the expression may be reduced on another processor. If there is no other processor available, it behaves like an `{| I |}`–annotation.
- `{| P at p |}`, with `p :: ProcId`: the expression is reduced on the processor identified by `p`. Every processor has a unique processor identifier of type `ProcId`, which can be obtained using one of the following system functions:
  - `currentProc` returns the processor identifier of the processor reducing it. `{| P at currentProc |}` behaves like `{| I |}`
  - `randomProc` returns the processor identifier of a randomly picked processor.
  - `toProcId x` transforms an integer into a processor identifier. All processors have an unique number, making it possible to pick a certain processor.

2.2 Communication

In the case of an `{| I |}`–like annotation, no communication is needed, as the expression will be reduced in the same heap and thus sharing can be used. When an annotated expression is sent to another processor, a channel node is created in the local heap. When this channel node is evaluated, a request is sent to the processor holding the expression. On the remote processor, a thread will evaluate the expression to root normal form and return the result for every request it receives. The graphs are copied between heaps using lazy normal form copying [Kes94], which avoids copying work: when an unevaluated node is reached, a channel node is inserted instead of copying it.

3 Why New Communication Primitives?

3.1 No Data-driven Communication

Inter-process communication is available in the form of implicit lazy copying, which fits naturally with the lazy evaluation strategy of Clean. The disadvantage is that it can be rather slow: every graph should be requested before it is sent, doubling the number of messages. Lazy copying is therefore similar to demand-driven communication. A more efficient form of communication is data-driven communication, where data is sent as soon as it becomes available. Concurrent Clean does not provide a direct way to express this. The only data-driven communication happens when a new reducer is created: the expression is sent away directly, without request. This cannot be used to implement general data-driven communication efficiently as with every message a new reducer is created, which gives a large overhead. The remote values presented in [Ser97a] use it to decrease communication overhead and give reasonable results, but real data-driven communication could improve these results a bit as no processes need to be created and collected.

3.2 No Non-determinism

Concurrent Clean is aimed to be an all-purpose programming language, so it should be possible to implement all kinds of systems with it. But as programs written with Concurrent Clean are deterministic, it is not possible to write a multi-user system with it, as such a system is inherently non-deterministic. If we want an all-purpose language, it should provide a way to express non-determinism.

3.3 No Inter-program Communication

There is not a real way to do communication with other programs, both Clean and non-Clean programs. Only files could be used, in a way like UNIX pipes, but on most systems this is not efficient.

We could provide an direct interface to a standard communication library, like TCP/IP, but then we would have two different systems for internal and external communication. It would be better if the same set of operations could be used for both internal and external communication.

Dynamic types [Pil97], which are currently being added to Clean, enable type checking objects which live longer than the program which uses them. An important use of dynamics is file-I/O: it enables us to type check the contents of files, but it can also be used for inter-program communication.

If we had inter-program communication, arbitrary graphs could be communicated, both data and functions of any kind. The runtime type checks make sure that the type of the message and the type which the receiver expects correspond. Without dynamic types, we could have safe inter-program communication, but only with messages of basic types, like integers and characters.

3.4 No Multicasting

Multicasting is getting more and more attention lately, especially for the internet. Broadcasting services, like audio and video on demand, send away a lot of data to a large number of destinations. It is almost obligatory to use multicasting here.

A multicast primitive is proposed for Clean [Ser97b]. It behaves similar as a `{| P |}`-annotation, so it always creates new reducers. It has therefore the same drawback: it's not very suitable for data-driven communication as a reducer is created with every message, resulting in more overhead and lower efficiency.

3.5 Hard to Use

It is possible to use lazy lists or streams for communication, but writing processes which have more than one input and output stream is cumbersome.

4 Known Solutions

4.1 Object I/O Message Passing

Clean's Object I/O system [AP97] already provides processes and message passing. The processes are built on top of Clean and the Object I/O does its own scheduling for this. Processes can send messages to receivers, which are special event handlers. Upon receiving a message, the receiver calls a handler which then uses the message to perform some actions.

Although the object I/O system's message passing has some of the properties we want, it does not have them all: it is deterministic and it does not give inter-program communication or multicasting. Moreover, it depends on the object I/O processes and event handlers to work. This last property makes it unsuitable for a general message passing system in Concurrent Clean.

However, if we have a communication mechanism in Concurrent Clean, it may be well possible to implement the object I/O message passing on top of it, as our requirements are stronger than what the object I/O requests.

4.2 Concurrent Haskell

Concurrent Haskell [PGF96] does not implement message passing directly, but bases it on mutable locations, called `MVar`'s. A value of type `MVar t` is a mutable location which can be empty or contain a single value of type `t`. Three operations are defined on `MVar`'s: `newMVar`, for creating a new `MVar`, `takeMVar`, which reads out an `MVar` and blocks when it is empty, and `putMVar` which is used to fill an `MVar`. It is an error when more than one process tries to fill an `MVar` at the same time. Non-determinism is introduced by the fact that when more than one processor waits for a value in an `MVar`, only one of them will get it. In the paper various examples are given for semaphores and channels. A drawback of this approach is that the communication is still demand-driven: no new communication mechanism is introduced, but the existing implicit communication mechanism is used.

4.3 CML, Facile, Scholz

Concurrent ML [Rep93] and Facile [TLP+93] provide similar synchronous communication primitives. In the former they are called `send` and `accept` and in the latter `send` and `receive`. A process can only send something when a matching receiving process is ready to receive. So both the sender and receiver will block when the other party is not available. As this protocol is strictly one-to-one, not multicasting is possible. The matching of senders and receivers is non-deterministic, a sender cannot determine which process will receive the message. This kind of non-determinism makes reasoning hard, as it is difficult hard to ensure that a message will arrive at the right place. Scholtz [Sch95] proposes a similar set of primitives for Haskell.

4.4 Jones and Hudak

A similar approach is taken by Jones and Hudak [JH93], which use the IO-monad to thread the communication actions. They also state that it is an error when two processors use the same channel when they are not a sender and a receiver. This ensures that no non-deterministic effects will take place. The approach proposed is therefore a deterministic one.

4.5 Reliable Multicast Transport Protocol (RMTP)

Reliable multicasting protocols have only recently received attention (from 1992 onwards). One recent protocol is the Reliable Multicast Transport Protocol (RMTP) [PKLB97]. It provides one-to-many communication which is reliable over wide area networks. This rather low-level mechanism can be used to implement more complex many-to-many protocols, including the Clean message passing system we propose in this paper.

In RMTP, a receiver will receive only messages sent from the moment it joins in, which makes reasoning harder. When all receivers should receive all messages, no matter when they join in, the sender should buffer all messages. RMTP provides for this a two-level data-cache. Only the most recent messages are cached in memory, the rest is stored on disk.

4.6 Totem, Isis, Transis

Next to this, quite some work has been done on ordered multicasting systems. In these systems, like Totem [MMSA+96], Isis [BvR94], Horus [vRBM96] and Transis [DM96], all messages on the same channel are ordered, even when they were sent by different senders. This implies that every receivers receives all messages in the same order, which makes reasoning easier, but has some overhead. It is achieved by putting the senders on a token ring. Senders are only allowed to send when they have the token. The token contains a sequence number, seq. Every sender having the token, may send its n messages, with message numbers seq to $seq+n-1$. The token is then passed to the next sender with value $seq+n$. In this way all message have a unique number, which can be used to order the messages.

5 Introducing Message Passing in Clean

Message passing is a form of I/O. Clean uses the world as values paradigm [Ach96], based on the uniqueness type system [BS93] [BS95]. An interactive Clean program is a function of type `*World -> *World`. `World` is an abstract, specialised type which represents the complete environment of the program. The uniqueness attribute `*` states that the environment type is unique, enabling destructive updates on the environment. The message passing functions will initially use the `World` environment for doing I/O.

5.1 Communication between Programs

We want to have an asynchronous message passing system. In such a system sending and receiving are not synchronised. In a data-driven communication system, the sender always should know the locations of the receivers. The messages sent to the receivers will be queued there, until the receiver accepts the message.

Our message passing system uses channels which are split in a sending and a receiving part: the send channel and the receive channel. Send channels as well as receive channels can be duplicated, enabling many-to-many channels. Sending can only happen on a send channel, while receiving is only allowed on receive channels. A message sent on a send channel will be send to all locations which possess the corresponding receive channels, where they will be queued until needed.

Programs may obtain channels using one of the following two functions:

```
:: SChannel a
:: RChannel a

:: InetHost :== String    // IP-address

:: Maybe a = Just a | Nothing

createRChannel :: String *World -> (RChannel a, *World)
lookupSChannel :: InetHost String *World ->
                                      (Maybe (SChannel a), *World)
```

`createRChannel` creates a channel and registers it under the given name. The receive side of the channel is then returned. We chose to return the receive channel, because in this way we can wait until some other program has looked up this channel by receiving on this receive channel. This is much harder if the returned channel was a send channel. The `lookupSChannel` function tries to find a channel under the given name on a machine with the given IP-address. If the channels exists, the function will return `Just` the channel, while it returns `Nothing` in the other case. Multiple programs may lookup the same channel, while it is possible to use `createrChannel` more than once with the same name, in case it will return the same RChannel. This gives us many-to-many communication.

Of course, dynamic typing is needed for inter-program communication (see [Pil97], but for simplicity we assume that the messages between programs can be typed statically.

Furthermore we have a number of functions on send and receive channels:

```
send      :: (SChannel a) a *World -> *World

receive   :: (RChannel a) *World -> (a, RChannel a, *World)
available :: (RChannel a) *World -> (Bool, *World)
```

As stated above, the function `send` sends a message to all receivers on the channel, thus the locations of all corresponding receive channels. The runtime system guarantees that all sent messages will be queued at all receivers in the same order as they are sent.

We noticed that the same channel can be looked up more than once, resulting in multiple send channels on the same channel. When these are independent, which is the case when multiple programs have the same send channel, the ordering between messages sent on those send channels is determined by a non-deterministic merge at the receivers (see Figure 1). This implies that the ordering between data sent by multiple independent sender may be different for every receiver.

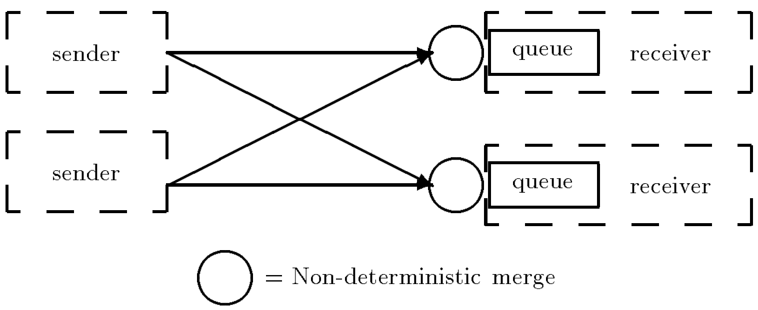


Fig. 1. message passing semantics

The `receive` function retrieves one message from the buffer. The rest of the messages are returned in the form of a new receive channel. This function is blocking, so when there are no messages available, the program will be blocked until one arrives. This blocking behaviour can be avoided using the function `available`, which returns a boolean stating whether a message is available for receiving.

We do not provide a function for closing channels. It is assumed that the garbage collector will close unused channels. A connection between two machines can be closed when either the send channel or the receive channel has become garbage.

In the next example, we have two programs: one being a producer, sending the numbers 1 to 10 on a send channel which has the name `"ConsChannel"` and will be looked up at the machine with the name `"consumer.net"`. The other program should run on that machine and consumes the messages on the channel it created under the name `"ConsChannel"` and sums the received numbers up:

(The keyword `#` indicates a let-expression which can be defined before a guard. It introduces a new lexical scope, while the right-hand-side identifiers can be re-used on the left-hand-side; they are internally tagged with a number)

```
// Program 1, the producer

Start :: *World -> *World
Start world
   #  (maybe_s_ch, world) =
        lookupSChannel "consumer.net" "ConsChannel" world
   = producer maybe_s_ch world
where
   producer :: (Maybe (SChannel Int)) *World -> *World
   producer Nothing     world = abort "channel not found"
   producer (Just s_ch) world = produce 1 s_ch world

produce :: Int (SChannel Int) *World -> *World
produce i s_ch world
   | i > 10
      = world
   | otherwise
      #  world = send s_ch i world
      = produce (i + 1) s_ch world

// Program 2, the consumer
// Should run on the machine with IP-address comsumer.net

Start :: *World -> (Int, *World)
Start world
   #  (r_ch, world) = createRChannel "ConsChannel" world
   = consume 0 r_ch world

consume :: Int (RChannel Int) *World -> (Int, *World)
consume r r_ch world
   #  (i, r_ch, world) = receive r_ch world
   | i == 10
      = (r + i, world)
   | otherwise
      = consume (r + i) r_ch world
```

5.2 Communication between Threads

In the example above we we have communication between two programs, but we would also like to have the same system of producers and consumers in one program, but this gives some troubles. As the producer and consumer are independent, we need independent sequences of computation inside one program. We will call these independent sequences 'threads'. Threads behave like programs but use thread states and have type `*TState -> *TState` instead of `*World -> *World`.

Threads can be created using the function `newThread` and its derivatives, which can be used with another `TState`, the world environment or the `PState l p` (for Object I/O programs).

```
class ThreadEnv e
where
   newThread      :: (*TState -> *TState) *e -> *e
   newIntThread   :: (*TState -> *TState) *e -> *e
   newParThread   :: (*TState -> *TState) *e -> *e
   newParThreadAt :: ProcId
                     (*TState -> *TState) *e -> *e

   newThread`      :: (*TState -> (a, *TState)) *e -> (a, *e)
   newIntThread`   :: (*TState -> (a, *TState)) *e -> (a, *e)
   newParThread`   :: (*TState -> (a, *TState)) *e -> (a, *e)
   newParThreadAt` :: ProcId
                      (*TState -> (a, *TState)) *e -> (a, *e)

instance ThreadEnv TState
instance ThreadEnv World
instance ThreadEnv (PState l p)
```

The `newThread` functions come in two flavours: one where the new thread only returns the final thread state and a primed version where an additional result value is returned. Next to this we have four evaluation strategies for threads:

`newThread` This thread is evaluated sequentially, `newThread` will only return the result environment when the thread has finished.

`newIntThread` A new interleaved reducer is created which will reduce the new thread concurrently with the parent thread. `newThread` therefore finishes immediately after the new reducer has been created.

`newParThread` Is similar to `newIntThread`, but uses a parallel reducer instead, so the thread may run on another processor.

`NewParThreadAt` Behaves the same as `newParThreadAt`, but has the extra feature that the processor which should be used can be chosen.

The last three versions correspond of course with the `{| I |}`, `{| P |}` and `{| P at p |}` annotations.

Channels for local use can be created using `newChannel`. It basically an efficient implementation of a `createRChannel`, followed by a `lookupSChannel` on the same machine using the same randomly chosen name. As this always succeeds, it gives you both the send and receive channels:

```
newChannel :: *TState -> (SChannel a, RChannel a, *TState)
```

Now we can write the producer-consumer program above using two threads instead of two independent programs:

```
Start :: *World -> (Int, *World)
Start world = newThread prodcons world
where
   prodcons :: *TState -> (Int, *TState)
   prodcons tState
      #  (s_ch, r_ch, tState)
                          = newChannel tState
         tState           = newParThread  (produce 1 s_ch) tState
         (r, tState)      = newParThread` (consume 0 r_ch) tState
      = (r, tState)

produce :: Int (SChannel Int) *TState -> *TState
produce i s_ch tState
   | i > 10
      = tState
   | otherwise
      #   tState = send s_ch i tState
      = produce (i + 1) s_ch tState

consume :: Int (RChannel Int) *TState -> *TState
consume r r_ch tState
   #  (i, r_ch, tState) = receive r_ch tState
   | i == 10
      = (r + i, tState)
   | otherwise
      = consume (r + i) r_ch tState
```

5.3 Communicating Unique Messages

One thing we have to decide is whether we want to communicate unique graphs. If we do not allow this, message passing can be done on a many-to-many basis: we do not have to restrict the number of senders or receivers. When we want to send unique graphs, we have to ensure only one receiver is able to receive such a graph. When this graph is copied to another heap, this does not seem harmful: we can have two unique references, each to one copy of the graph. But it is generally undecidable whether a graph will be copied to another heap or shared in the same heap, so we decided to include this restriction.

Sending unique graphs maybe useful: one could imagine two threads within one program running on the same processor. Both threads manipulate a datastructure in turn. When a thread has finished with the datastructure it will send it (with an uniqueness attribute) to the other thread, which can then read the changes and manipulate the datastructure.

Another setting where sending unique graphs is useful is in token situations: a token is used by a number of threads to denote a privilege. A thread having the token is allowed to perform some action, like manipulating a data-structure or sending a message to a conference, where many other threads can see that message and threads may 'speak' in turn. When the token is made unique, ensuring that there is only one keeper of the token at all times is done by the type system and thus trivial. Of course the uniqueness property of the token should then be conserved while sending it over a channel.

The channels introduced in the previous subsection are not suitable for sending unique messages, because we can not prevent that the same message is received by several receivers and thereby possibly creating two references to that message.

To enable unique messages, we should restrict the number of receivers to exactly one when an unique message is sent. However, when we send non-unique graphs, there is no need to restrict the number of receivers. So we have to look for a mechanism that restricts the number of receivers to one, only when unique messages are sent over that channel.

We can use the uniqueness typing system to ensure that only one reference to a receive channel exists. Uniqueness type variables can be used to ensure that a receive channel is only unique when the messages it receives are unique. At the sender side, there is no restriction, so a send channel may or may not be unique when unique graphs are sent across.

```
newChannel :: *TState ->
           (.(SChannel u:a), v:(RChannel u:a), *TState), [v <= u]
```

We see from the type that when the uniqueness variable `u` is instantiated with the unicity property, the uniqueness variable `v` also should be unique, because of the restriction `[v <= u]`. So when the receiver side of the channel receives unique graphs, the receive channel has to be unique too. The send channel does not have such a restriction.

Because of the unicity properties of the new channels, we have to alter the types of the send and receive functions. The tree primitive function now preserve the unicity of channels, while the `send` and `available` functions can be derived from the `u_send` and `u_available` functions by throwing away the result channel.

```
u_send      :: u:(SChannel .a) .a *TState ->
                                     (u:(SChannel .a), *TState)

receive     :: u:(RChannel .a) *TState ->
                                (.a,   u:(RChannel .a), *TState)
```

```
u_available :: u:(RChannel .a) *TState ->
                                 (Bool, u:(RChannel .a), *TState)

send :: .(SChannel .a) .a *TState -> *TState
send ch msg tState = snd (u_send ch msg tState)

available :: .(RChannel .a) *TState -> (Bool, *TState)
available ch tState = snd (u_available ch tState)
```

We also notice that send actions are now independent when they do not use the same send channel and thread state.

Now it is possible to write a token-game function, where the uniqueness of the token is guaranteed by the type system. In the example below, the function `tokenGame` is a function in a ring of similar functions, connected by channels. Every function has a channel to the next in the ring. The receive side of this channel is unique, because the token is unique. When the token is received from the previous function, a function `f` is applied to it. Then the token is send to the next function in the ring and this function itself restarts, waiting on the token to arrive again.

```
tokenGame :: *(RChannel *Token) (SChannel *Token)
                            (*Token -> *Token) *TState -> *TState
tokenGame from_prev to_next f tState
   #   (token, from_prev, tState) = receive from_prev tState
       token                      = f token
       tState                     = send to_next token tState
   = tokenGame from_prev to_next f tState
```

5.4 An example: A Binary Semaphore

As a final example we show an implementation for semaphores. Semaphores are used to ensure safe access to critical regions. We can implement a behaviour like that using a thread handling the requests for entering and leaving. This thread has two channels, one for the entering requests sent by `wait` and one for the leave messages sent by `signal`.

The `wait` function creates a new channel which is used for the reply to the message it sends on the `enter_ch` channel. When it receives a reply it may safely proceed into the critical region. The contents of the reply are not needed and is therefore set to a dummy value.

The `signal` function only sends a dummy message on the `leave_ch` channel, which will be used by the server to proceed to the next waiting process.

The `server` ensures that at most one thread is in the critical region. Initially no thread is in the critical region, so we can allow one thread to enter. Therefore we receive an request on the `enter_rc` channel and return dummy reply on the received reply channel. As `wait` blocks until it receives a reply permission, that thread will then enter the critical region. Now the semaphore process blocks until it receives a dummy message on the `leave_rc` channel from the thread coming out of the critical region. As soon as that has arrived, we know that the critical region is free again.

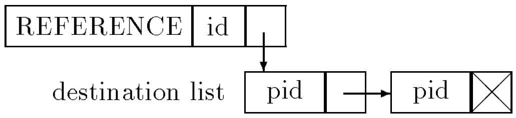
```
:: Dummy = Dummy


:: Semaphore = {
      enter_ch :: !SChannel (SChannel Dummy),
      leave_ch :: !SChannel Dummy
      }


newSemaphore :: *TState -> (Semaphore, *TState)
newSemaphore tState
   #   (enter_sc, enter_rc, tState) = newChannel tState
       (leave_sc, leave_rc, tState) = newChannel tState

       tState    = newIntThread (server enter_rc leave_rc) tState

       semaphore = {
          enter_ch = enter_sc,
          leave_ch = leave_sc
          }
   = (semaphore, tState)
where
   server :: (RChannel (SChannel Dummy)) (RChannel Dummy)
                                                  *TState -> *TState
   server enter_rc leave_rc tState
      #   (reply_ch, enter_rc, tState)
                                = receive enter_rc tState

          tState                = send reply_ch Dummy tState
          (_, leave_rc, tState) = receive leave_rc tState
      = server enter_rc leave_rc tState

wait :: Semaphore *TState -> *TState
wait semaphore tState
   #   (reply_sc, reply_rc, tState) = newChannel tState

       tState      = send semaphore.enter_ch reply_sc tState
       (_, _, mps) = receive reply_rc tState
   = tState

signal :: Semaphore *TState -> *TState
signal semaphore tState
```

```
= send semaphore.leave_ch Dummy tState
```

6 Implementation

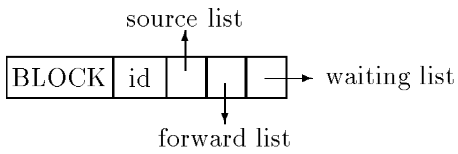
6.1 New Graph Nodes

In the current implementation, the Clean send and receive channels from Section 5.1 are represented in the heap by reference and block nodes respectively. References point to zero or more block nodes. Every message channel has an unique global identification, which is used by the reference nodes to determine the block nodes they are referring to.

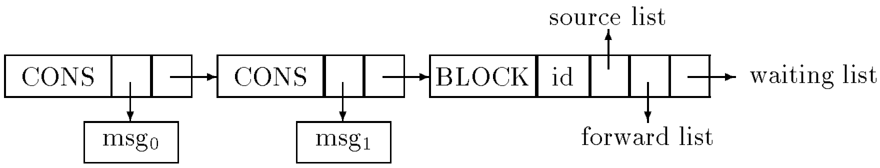


References nodes contain, apart from the identification, a destination list. Messages are send to all locations in the destination list of the reference node, which are the locations of the block nodes. The messages are copied between heaps using the lazy normal form copying [Kes94]. This technique avoids copying work: when an unevaluated node is met during copying, a channel node is inserted in the graph copy.

On the other side we have block nodes, which are similar to channel nodes:



When a block node is evaluated by a reducer, the reducer will be placed in the waiting list of the block node. In contrast to channel nodes, no request message is sent. When a message arrives for the block node, the node is overwritten by a `Cons` node with as arguments the arrived message and a reference to a new block node, with the same identification as the old one. When the next message arrives, this new block node will be overwritten. So if two messages have been sent to the block node above, we will have to following graph in the heap:



The forward list is used when a block node is copied: the location of the copy is stored in the forward list of the original. When a message arrives, it is forwarded automatically to all locations in the forward list. A location is removed from the forward list, when the messages are sent to the forward location directly and thus forwarding is not needed anymore.

The source list is a list with the locations of all references to this block node. It is used to suspend sending messages in certain situations (see Section 6.2).

Block and references nodes are very flexible: many reference nodes can point to the same block node (this is needed for a non-deterministic merge) and a reference node can point to many block nodes (for multicasting).

6.2 Message Ordering

For having a system where non-determinism is only introduced by multiple senders, we should guarantee that all messages sent by a certain sender arrive in the right order at all receivers.

During normal communication, there are no problems, because we assume a reliable communications protocol. However, when a receiver is copied to another processor a problem may arise. When a receiver is copied, the message queue may contain messages which have not yet been processed by the receiver. These messages will not be sent to the new receiver location directly. As soon as the receiver processes these messages, they will be forwarded to the new receiver. The messages sent after copying the receiver will be sent to both receivers directly, so there is no need to forward these.

Now suppose that the first receiver crashes between making the new copy and forwarding the unprocessed messages. The new receiver will not receive the unprocessed messages, but will receive the messages sent after the copying. So in that case the unprocessed messages are lost. We will present two solutions to this problem, of which the second has been implemented in the current parallel runtime system.

Message Numbering The first solution is to enumerate all messages. All receivers will compare the number of the incoming messages with the the requested message number, which is the number of the last message which was received correctly incremented by one. If:

The new message number is less This means that this message has been received already and this message can be thrown away.

The new message number is equal The message is the correct message and will therefore be put in the message queue, the requested message number will be incremented.

The new message number is higher This says that the right next message has not been received yet. We put this message in a buffer, while continue waiting for the correct message. A buffered message will be put in the queue when the requested message number equals its message number.

This mechanism guarantees a total ordering of the messages. In the problem case above, the new receiver got a message from the sender before receiving the forwarded message from the first receiver, because the latter crashed. With the message numbering, the new receiver will continue to wait for the forwarded message if it receives the messages from the sender, because the message numbers of the messages coming from the sender are higher than the numbers of the forwarded messages.

The advantage of this mechanism is that the sender does not have to stop sending, so other receivers will not notice anything of the copying of the receiver. The disadvantage is that the buffer storing out-of-sequence messages will eventually run out of memory if the first receiver has crashed. The new receiver will never put the messages from the buffer in the queue, because the forwarded messages will never arrive. A second disadvantage is that every sender on the same channel uses a separate numbering, as there is no synchronisation between senders. This complicates the implementation at the receiver side, as it has to keep track of message numbers for each sender separately.

Interrupting Sending The second solution is based on the observation that when the problem occurs, the first messages sent by the sender to the new receiver arrive before the forwarded messages from the first receiver. The solution based upon this therefore divides the messages into two groups, before and after copying the receiver and makes sure that these two groups of messages are sent in the right order.

When a receiver is copied, a message is sent to all senders. This message also divides the message into the two groups mentioned above. The first group are the message sent before the message to the sender arrived, while the second group consists of the messages after that moment. Now we have to guarantee two things: the messages of each group should all arrive in the right order and all messages of the first group should arrive before any message of the second group. The total ordering of the messages in the first group is ensured by the fact that the first receiver receives them in the correct order (as this is normal communication, see above), while this order is preserved when forwarding them to the new receiver. The total ordering in the second group is trivial, as it is normal communication.

Making sure that all messages of the first group arrive before any of the second group can be achieved by delaying sending the messages from the second group until all messages of the first group have arrived at the new receiver. This can be done by making the message to the sender which divides the two groups into a stop message: the sender will stop sending message when it receives the message. The sender will send an acknowledge reply to the receiver. The first receiver can now determine the messages belonging to the first group, namely all processed messages plus the unprocessed messages in the message queue. The first receiver will then send all these messages in the right order to the new receiver. When the new receiver has received all these messages we know that all messages of group one have arrived before those of group two. The last action then is that the new receiver sends a message to the sender for starting the sending of the messages from the second group. Where the previous solution had the possibility for a space leak, in this case a larger part of the program will block indefinitely when the forwarding processor crashes, because the sender will not send message any more.

As the sender has to wait sending for a certain period, other receivers may notice a short pause in the sending of the messages. This may give problems in the case where the continuity of the message stream is critical, but as the garbage collector can cause a similar pause, we do not think this is a major problem.

7 Current Work

7.1 More Message Ordering

In Section 5.1 it was noted that send actions are only dependent when they use the same channel, environment and heap. But sometimes this is too limiting.

The message order is not always satisfying with multiple senders and receivers. According to the semantic model, the ordering between messages from different senders is undefined, but can also be different for every receiver. This makes reasoning about this system harder, a receiver getting messages in a certain order cannot assume that other receivers on the same channel received the messages in the same order. We already saw that this was avoided in systems like Totem, Isis and Transis. They use a token ring with the number of the next message to be send as token. A send action can only take place when it has the token and sends the message with the number from the token, after which the message number is increased and the token is send onwards. We could use a similar technique, but there is a better solution.

In this solution we also have a token containing the message number of the next message to be send, but it does not travel over a ring. There is always one sender holding the token. When a send action takes place, a token request is multicasted to all other senders. The sender that holds the token will then return it. When the token is received, the send action takes place in the same way as for the token ring solution. This solution has the advantage that there is no token traveling around when there is nothing to send and thus is more efficient when there is not much to send.

We could include this token game in the primitive channels, but this introduces quite some overhead in all channels, while not all channels need it, only some with multiple senders and receivers. The best solution seems to be to implement this token game on top of our channels and provide it as a special channel, while overloading the send and receive functions, so that it can be used as any other channels. In the near future we expect to see more of these special purpose channels.

7.2 Concurrent Object I/O

We are currently working on a concurrent implementation of the object I/O system. In this system, every interactive process is implemented using the threads from Concurrent Clean. The receivers are implemented using the message passing primitives as they are introduced in this paper. By doing this, we are able to remove a substantial part of the object I/O program code which dealt with scheduling and receivers. Moreover, we expect that the performance will improve, as the primitive threads and message passing are more efficient than those implemented in the current sequential object I/O system.

8 Conclusions

We noticed that Concurrent Clean had some drawbacks with respect to concurrency. Multicasting and data-driven communication were not efficient, while non-determinism and inter-program communication were not possible. Furthermore we noticed that lazy lists were not really suited for flexible communication.

An extension for Concurrent Clean was proposed providing flexible message passing. The channels used for communication are split in a send and a receive part, which clarifies the direction in which messages are sent, while it provided us the possibility to restrict the number of receivers to exactly one for when unique messages are sent.

The channels can be used for communication between programs, but also within one program. For the latter we needed independent threads with their own state, which behave similar to programs operating on the world. The possibility to evaluate these threads sequentially, interleaved or parallel gives us all we need to write real concurrent programs.

References

[Ach96] P.M. Achten. *Interactive Functional Programs - Models, Methods and Implementation.* PhD thesis, University of Nijmegen, February 1996.

[AP97] P.M. Achten and M.J. Plasmeijer. Interactive functional objects in Clean. In Clack et al. [CDH97], pages 305–322.

[BS93] E. Barendsen and J.E.W. Smetsers. Conventional and uniqueness typing in graph rewrite systems. In R.K. Shyamasundar, editor, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *LNCS*, pages 41–51. Springer-Verlag, 1993. extented abstract.

[BS95] E. Barendsen and J.E.W. Smetsers. Uniqueness type inference. In Hermenegildo and Swierstra, editors, *Proceedings of Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982 of *LNCS*, pages 189–207. Springer-Verlag, 1995.

[BvR94] K.P. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, Los Alamitos, CA, 1994.

[CDH97] C. Clack, A.J.T. Davie, and K. Hammond, editors. *Implementation of Functional Languages, 9th International Workshop, Selected Papers*, volume 1467 of *LNCS*. Springer-Verlag, September 1997.

[DM96] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39, Vol. 4:64–70, April 1996.

[HC93] I. Holyer and D. Carter. Deterministic concurrency. In *Proceedings of the 1993 Glasgow Workshop on Functional Programming*, pages 113–126. Springer-Verlag, July 1993.

[JH93] M.P. Jones and P. Hudak. Implicit and explicit parallel programming in Haskell. Technical Report YALEU/DCS/RR-982, Yale University, 1993.

[Kes94] M.H.G. Kesseler. Uniqueness and lazy graph copying - copyright for the unique. In *Proceedings of the 6th International Workshop on the Implementation of Functional Languages*, Norwich, UK, 1994. University of East Anglia.

[MMSA+96] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, Budhia R.K., and C.C. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39, Vol. 4:54–63, April 1996.

[NSvEP91] E.G.J.M.H. Nöcker, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J. Plasmeijer. Concurrent Clean. In *Proceedings of Parallel Architectures and Languages Europe*, number 506 in LNCS, pages 202–219, Eindhoven, The Netherlands, June 1991. Spinger-Verlag.

[PGF96] S.L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages (POPL '96)*, St Petersburg Beach, Florida, January 1996.

[Pil97] M.R.C. Pil. First class file I/O. In Werner Kluge, editor, *Implementation of Functional Languages, 8th International Workshop, Selected Papers*, volume 1268 of *LNCS*, pages 233–246, 1997.

[PKLB97] S. Paul, Sabnani K.K, J.C. Lin, and S. Bhattacharyya. Reliable multicast transport protocol (RMTP). *IEEE Journal on Selected Areas in Communications*, 15(3):407–421, April 1997.

[Rep93] J. H. Reppy. Concurrent ML: Design, application and semantics. In P. E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198. Springer, Berlin, Heidelberg, 1993.

[Sch95] E. Scholz. Four concurrency primitives for Haskell. In *ACM/IFIP Haskell Workshop*, La Jolla, Colifornia, 1995. Research Report YALEU/DCS/RR-1075.

[Ser97a] P.R. Serrarens. Distributed arrays in the functional language Concurrent Clean. In *Proceedings of the 3rd international Euro-Par conference*, volume 1300 of *LNCS*, pages 1201–1208, Passau, Germany, August 1997.

[Ser97b] P.R. Serrarens. Using multicasting for optimising data-parallelism. In Clack et al. [CDH97], pages 271–285.

[TLP+93] B. Thomsen, L. Leth, S. Prasad, T-M. Kuo, A. Kramer, F. Knabe, and A. Giacalone. Facile antigua release programming guide. Technical Report ECRC-93-20, European Computer-Industry Research Centre, 1993.

[Tur95] D.N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, Edinburgh University, 1995.

[vRBM96] R. van Renesse, K.P. Birman, and S. Maffeis. Horus: A flexible group communications system. *Communications of the ACM*, 39, Vol. 4, April 1996.