

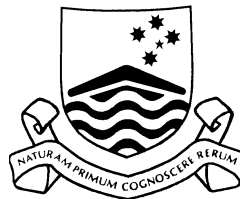
ANU/Fujitsu CAP Research Program

AP1000 Software Release 1994

paraML

Programming Manual

*Department of Computer Science,
Computer Sciences Laboratory,
The Australian National University*



Centre for Information Science Research

All Rights Reserved. Copyright ©1993,1994 The Australian National University. This document is published in accordance with the Fujitsu-ANU joint research agreement. No part of this publication may be reproduced without permission of the ANU CAP Advisory Committee and Fujitsu Laboratories Ltd.

Contents

1. Introduction	1
2. Programming Model as an Abstract Machine	1
3. Language Primitives	2
3.1 Processes in paraML	2
3.1.1 Process Definition and Creation	2
3.1.2 Creation of Process Groups	3
3.1.3 Results from Processes	3
3.2 Communication	4
3.2.1 Fixed Port Creation	4
3.2.2 Dynamic Ports	5
3.2.3 Retrieving Port Names from Fixed Ports	5
3.2.4 Message Passing	5
3.3 Identity	8
3.4 Program Execution	8
3.5 Concurrent ML	9
3.6 Miscellaneous	9
3.6.1 Printing	9
3.6.2 Configuration Information	10
3.6.3 Environment Information	10
3.6.4 Process Scheduling	10
4. Compiling and Running	10
Appendix A – Signature File for the ParaML Structure	12
References	13

1. Introduction

This document provides an introductory programming manual for paraML, an extension of ML. The classic example of computing factorial motivates the explanation of paraML syntax, what can and cannot be done, and how to work around some of problems that arise through the design of the language and its interaction with the type system.

There is no attempt made in this report to justify the overall design for paraML; for more information about our decisions and motivation in that regard, please see [3,4,5,6,9]. Justifications will be made about particular primitives and how they are implemented, since we chose to make no change to the syntax of Standard ML, and simply provide the extensions as an ML module, in the same manner as Concurrent ML[11,12].

The current implementation has been built for the Fujitsu AP1000, a distributed memory multicomputer. The primary modifications made to the compilation system (SML2C or SML/NJ) were to extend the runtime system to enable access to the AP1000's task creation and interprocessor communication primitives. I see no great difficulty in porting the system to the CM5, or similar multicomputers; it would require rewriting the runtime system calls. Similarly, the language could be ported to a sequential computer or a shared memory multiprocessor. The paraML operations could be rewritten in the latter cases with CML primitives, instead of accessing the runtime system communication operations. A similar approach could be taken to rewrite the paraML runtime system which executes on each processing cell of the multicomputer in CML.

Readers are expected to be familiar with the basic syntax of Standard ML as we make no attempt to describe the language in this manual. Excellent introductions to the language can be found in [10,14]. The formal definition of the language and a commentary are given in [7,8]. Lastly, since the system is built on top of Standard ML of New Jersey-based compilers, the facilities available under those systems are available in paraML; documentation on the New Jersey systems is available in [1].

2. Programming Model as an Abstract Machine

- A paraML program executes on a virtual machine consisting of arbitrarily many virtual processors; each physical processor has its own paraML runtime system capable of supporting multiple virtual processors. Until processes are created, the program behaves like a sequential ML program, executing on a single processor. This program is the top-level user expression.
- Each virtual processor manages its own independent environment. When a new process is created, it is assigned to some new virtual processor.
- Processes are created through expression evaluation; a name that uniquely identifies the new process is the value returned. The new process evaluates an expression in an initial environment which is a copy of the creator's at the time of creation.
- Communication between processes is performed by sending objects to typed ports. These ports are private and readable only by the process that owns them; they are not expressible objects, and do not exist as part of the local environment. Bindings to objects are made by reading a message into the local environment from a port. Ports come into existence on process creation, or may be dynamically created by a process during execution.

The new *semantic objects* in a paraML program are thus *processes*, which encapsulate an expression evaluating independently of other expressions, and *ports*, which queue messages (first-class ML objects) destined for a particular process. The new expressible *values* that permit access to these semantic objects are *process names* and *port names*. Both process names and port names are strongly typed, which permits static checking of communication between processes, though the communication may be non-deterministic.

3. Language Primitives

In this section, the various primitives of the paraML extension are introduced, grouped together where appropriate. A listing of the signature file for the paraML structure is given in Appendix A. The basic model of paraML programs is a user expression that is evaluated

3.1 Processes in paraML

Control parallel programming requires a minimum set of primitives to provide process creation and to enable interprocess cooperation. In paraML there are mechanisms for forming process declarations (without executing them), and to create actively-executing processes as instances of such process forms. A paraML operation can be used to ascertain the result of a process and message passing primitives are provided for communication between processes. In this section, just the mechanisms for declaring processes, creating them, and obtaining their result are illustrated.

3.1.1 Process Definition and Creation

Processes are defined typically with a function definition. An example definition for a process that computes factorial follows below:

```
fun factorial (n:int) = (create (<ports declaration>:'lp Ports)
                             (fn () => <process body>:'r))
```

For the moment, ignore what goes in the areas marked *<port declarations>* and *<process body>*. The function `factorial`, when applied to an integer argument, calls the `create` function. This function is in curried form, and takes a ports declaration (defined in Section 3.2.1), and a thunk (a function with unit argument). The thunk encodes the process body – the expression which will be evaluated by the process when executing. The unit function prevents the process body being evaluated immediately; instead, the thunk will be applied to the unit argument in the newly created process, thereby achieving parallel execution. Bindings to variables used in the process body occur in the normal manner with functions in ML. In particular, when `factorial` is applied, the value of `n` is bound within the process body. Importantly, reference variables and other mutable objects (like arrays) used in the process body, whose declaration occurs outside the process definition, are *copied* at application time – there is no shared global address space provided by paraML.

The value returned on applying `factorial` to an integer argument is an object of type `('r, 'lp) ProcessName`.

```
val fact_proc_name:('r,'lp) ProcessName = factorial 10
```

The process name encapsulates information about the type of the process body (`'r`), and the type of the communication ports (`'p`). It also encodes sufficient information for operations to retrieve

the communication ports, and to obtain the value computed by the process on evaluation of its process body. The type of the `create` operation is thus given by:

```
val create : 'lp Ports -> (unit -> 'r) -> ('r,'lp) ProcessName
```

3.1.2 Creation of Process Groups

It is also possible to create multiple instances of a process definition in a single operation. Suppose we wanted to create multiple factorial processes (though there is no particular reason why we would do this), the declaration for factorial would look like:

```
fun factorial (n:int) = (groupcreate (<ports declaration>:'lp Ports)
                                (fn () => <process body>:'r))
```

The `groupcreate` operation behaves almost exactly like the `create` operation, except that it expects an extra argument (again in curried form), which specifies the number of processes to be created. Instead of just a single process name being returned, a list of process names is returned. This list is, in a strong sense, a *group* of processes, for reasons that will be explained later in Section 3.2.4. The process creation now looks like this:

```
val fact_proc_names:( 'r,'lp) ProcessName list = factorial 10 5
```

The important thing to note is that each process is unique, and has a different process name, but that they will contain identical bindings of variables passed as arguments to the function application and variable declared outside of the process definition within their process body. The type of the `groupcreate` operation is given as:

```
val groupcreate : 'lc Ports -> (unit -> 'r) -> int ->
                  ('r,'lc) ProcessName list
```

3.1.3 Results from Processes

Once a process has been created, there are various ways to communicate with it using message passing. If it is necessary to synchronise on a process's termination, or to obtain the value computed by a process's body, the `result` operation may be used. The function `result` takes a process name, and blocks until the process identified by the name has completed execution. The value computed by the process's body is returned as the value of the `result` function.

```
val fact10 = result fact_proc_name
```

Note that any process which knows another process's name may use the `result` operation; it is not the case that `result` can only be applied once for any given process name. The value computed by the process will be sent to any process that requests the process's result, and this value is only available after process termination. The type of `result` is:

```
val result : ('r,'lp) ProcessName -> 'r
```

Factorial

A complete definition of factorial can be written, which does not require communication. This example illustrates the use of recursive process creation. Ignore the ports declaration for the time being.

```

fun factorial (n:int) = (create (no_ports())
  (fn () =>
    if n = 0
    then 1
    else
      let val execproc = factorial (n-1)
          val part_fac = result execproc
          in n * part_fac
          end
      end))

```

3.2 Communication

Communication in paraML occurs via message passing to typed ports. Ports can come into existence either at process creation time (fixed ports), or throughout the evaluation of a process body (dynamic ports). There exist various operations to manipulate messages and ports. Any first class ML object may be communicated to an appropriately-typed port, including functions, arrays, records, lists, process names etc.

3.2.1 Fixed Port Creation

It is useful to be able to declare ports that will come into existence on process creation. Since these ports are declared in the process definition, and form part of a process's name, they are considered to be fixed. The operation to create a port is called, naturally enough, `port`, and takes a unit argument. The value returned by the `port` operation is a port name, of type `'1p PortName`. The type `'1p` indicates that this operation is a weakly-polymorphic¹ operation of rank 1. Essentially, what this requires is that at compilation time, all port name objects must have a ground type variable (such as `int` or `FooBar.data`) parameterising the `PortName` type. Typically, we wish to declare a number of ports, and to be able to identify them individually, which is most simply done using an ML record. The record is wrapped in a datatype constructor to indicate that it is a set of fixed ports for the `create` or `groupcreate` operations using the `ports` operation. For instance, the appropriate ports declaration to be used with the `factorial` process definition (given above in Section 3.1.1) might look like this:

```

type fact_ports = {data:int PortName}
  (ports ({data=port()}:fact_ports))

```

A single port only is required, which is declared within the ML record `{data=port()}`. The type of the port is given by the `fact_ports` type declaration, which determines the port is to receive integers. The field name `data` is thus bound to a port name object of type `int PortName`. The expression as a whole yields an object of type `fact_ports Ports`, which is then of the appropriate type to be used as the first argument to a `create` or `groupcreate` operation. The types of the `port` and `ports` operations are as follows:

```

val port : unit -> '1p PortName
val ports : '1a -> '1a Ports

```

1. Weak polymorphism is a mechanism used by the Standard ML of New Jersey compilers to eliminate type loopholes. A brief explanation of it is that the rank of the type variable indicates the number of abstractions that “protect” `PortName` values of that type.

In the case where there are no ports to declare on process creation (as in the earlier definition of factorial), the operation `no_ports` is provided. The operation returns a value of type `unit Ports`:

```
val no_ports : unit -> unit Ports
```

The code for the factorial process definition now looks like this:

```
type fact_ports = {data:int PortName}
fun factorial () = (create (ports ({data=port()}:fact_ports))
                    (fn () => <process body>:'r))
val fact_proc_name:( 'r, fact_ports) ProcessName = factorial ()
```

3.2.2 Dynamic Ports

Ports can be declared at any time, either within an executing process, or by the top-level evaluation which declares and creates the first processes. A port is declared using the `port` operation as before:

```
val dynamic_port_name:bool PortName = port()
```

It makes no difference to message passing operations whether the port name was generated from a fixed port declaration or a dynamic port declaration.

3.2.3 Retrieving Port Names from Fixed Ports

Since fixed ports are declared with the process definition, there must also be a mechanism to retrieve them from within the process name of a created process. The operation provided to do this is called `getports`, which takes a process name as argument, and returns the associated fixed ports. For example, retrieving the fixed ports from the factorial process would be possible with:

```
val ports_of_fact:fact_ports = getports fact_proc_name
```

Note that this retrieves an object of type `fact_ports`, not `fact_ports Ports`, since we now wish to be able to identify individual port names. The port name bound to `data` can be picked out with the ML record field selector operation `#`:

```
val fact_data_port_name = #data ports_of_fact
```

Within a process body, we wish to be able to do the same thing, and an operation called `selfports` is provided that retrieves the fixed ports associated with the process's name:

```
val ports_of_fact:fact_ports = selfports fact_proc_name
```

Using `selfports`, it is necessary to identify the type of the resulting object (`fact_ports` in this example), since `selfports` returns a weakly polymorphic value. Note that the `selfports` operation is equivalent to `getports (selfid())` – the `selfid` operation is defined in Section 3.3.

3.2.4 Message Passing

Sending

In order to send an object to a port, the operation `send` is provided which takes an argument of type `'lp PortName` and an object of type `'lp`, in curried form. Suppose we wish to send the value 9 to the data port of a factorial process, then the following performs this operation:

```
send fact_data_port_name 9
```

Note that sending is asynchronous, so there is no blocking for message receipt. The type of the send operation is clearly:

```
val send : 'lp PortName -> 'lp -> unit
```

It is also possible to broadcast messages to a fixed port of all processes belonging to a group (ie. those created by a `groupcreate` operation). The operation accepts the port name of any member of a process group, and broadcasts it to the same port of all processes.² If a member of a process group is broadcasting to a fixed port common to all members of its own group, then the message is not received by the sending process. Broadcast does not work for dynamic ports, or indeed for any port other than a fixed port of a `groupcreate` operation. The type of the `broad` operation is:

```
val broad : 'lp PortName -> 'lp -> unit
```

Message arrival order is preserved between successive message passing operations from one process to another, regardless of whether `send` or `broad` is used.

Receiving

In order to extract an object from a port, the operation `get` is provided, which takes a single port name argument (of type `'lp PortName`) and returns the first object from the port (of type `'lp`). Messages are queued in a port in order of arrival. Retrieving a message from the data port of the factorial process can be done by:

```
val data = get fact_data_port_name
```

The operation will block until a message is available.³ If the port name that is passed to `get` does not identify a port that is owned by the process which calls `get`, then an exception `PortNotOwned` is raised. The implication is that ports are not shared data structures that can be examined by any process which knows the port name – extracting messages can only be done by the process that creates a port, either as part of its fixed ports on creation, or a dynamic port created during execution. The type of the `get` operation is:

```
val get : 'lp PortName -> 'lp
```

Checking and Selecting

Since `get` is a blocking operation, it is also desirable to be able to check whether a message is actually ready to be extracted. The function `ready` takes a port name, and returns a function. The return function, when applied to the unit argument, yields a boolean value indicating whether there is a message available or not on the port. To check whether a message has arrived on the data port of the factorial process, we might do the following:

```
let val check_data = ready fact_data_port_name
in check_data ()
end
```

-
2. It is anticipated that the current mechanisms for broadcasting will change in the near future with the introduction of a `groupport` operation; `broad` will then only work with `GroupPortName` objects.
 3. If the operation does block, then the process also yields control to either another process or the paraML runtime system executing on each cell. At a later time, it will be swapped back in, and so on, until the message is available.

The type of the `ready` operation is thus:

```
val ready : 'lp PortName -> (unit -> bool)
```

More generally, an ability to select an action according to whether there is a message available on one of a number of ports is desirable. This operation is called `select`, and takes a list of tuples, where the first tuple element is a check function of type `unit -> bool`, and the second element is an action function of type `unit -> 'a`. The semantics of the operation are that the list is worked through progressively until one of the check functions returns `true`, and the corresponding action function is executed. If none of the check functions evaluates to `true`, it returns to the start of the list and tries again, and so forth until an action function can be executed.⁴ Suppose we have another port name called `term_port_name` of type `bool PortName`, and we want to check whether we have either data or a termination message, then we might do the following:

```
select [
  (ready fact_data_port_name, fn () => (<do something with data>;())) ),
  (ready term_port_name, fn () => (<do something on term>;())) ]
```

Things to note about `select` are that all the action functions must be of the same type, so that the return value of the whole operation is a value of the result type of the action functions. Since there is no guarantee of when a message will arrive, it is possible that between executing the first check function and executing the second check function, two messages will arrive, the first of which would have satisfied the first check function, and the second satisfies the second check function. Action functions should not assume that if their check function was satisfied that it implies that previous check functions could not be satisfied now, only that they were not satisfied when checked in their turn.⁵

The type of the `select` operation is:

```
val select : ((unit -> bool) * (unit -> 'a)) list -> 'a
```

Factorial Again

We now can rewrite `factorial` to use message communication, rather than passing the integer argument at creation time. The definition might look something like this:

```
type fact_ports = {data:int PortName}
fun factorial () =
  (create (ports ({data=port()}:fact_ports))
  (fn () =>
    let val my_ports:fact_ports = selfports()
        val fact_data_port_name = #data my_ports
        val data = get fact_data_port_name
    in if data = 0
       then 1
       else
```

4. If all check functions fail, then this is taken as an opportunity to perform a process switch, as with `get`.

5. It is hoped that a future version of `paraML` will implement a mechanism to read all messages each time through a checking iteration of the `select` loop, thereby guaranteeing action functions only execute if all previous check functions failed and would fail now.

```

    let val execproc = factorial ()
        val _ = send (#data ((getports execproc):fact_ports))
                    (execdata - 1)
        val part_fac = result execproc
    in data * part_fac
    end
end)

```

3.3 Identity

The ability for a process to obtain its own process name is provided with the operation `selfid`. This function takes a `unit` argument, and returns a process name. The process name object returned is weakly polymorphic, of type `('r, 'lp) ProcessName`, so care needs to be taken with manipulating it. The type of `selfid` is given as:

```
val selfid : unit -> ('r, 'lc) ProcessName
```

There are two functions which return unique integer ids, one for the process number and one for the group number. These are both evaluated within the context of a particular process. The first operation, `process_number`, returns the integer process id associated with the calling process, and is unique among all other processes in the system, starting from 1. The value 0 is returned if `process_number` is executed by the top-level user expression. The second operation, `group_number`, returns the integer group id associated with the calling process relative to other processes created in a `groupcreate` operation. Thus if n processes are created by a call to `groupcreate`, then these processes will return values from 0 to $n-1$, which correspond to the ordering of process names within the list returned by the `groupcreate` operation. If `group_number` is executed by a process created by a `create` operation, the value 0 is returned.⁶ The types of the `process_number` and `group_number` operations are as follows:

```

val process_number : unit -> int
val group_number : unit -> int

```

3.4 Program Execution

The current mechanism for compiling programs is based on a separate compilation system using a modified version of the SML2C compiler[13]. Signatures, functors, and structures are compiled to produce a single executable and the structures are executed for side effects. In this sense, all cells execute the same code, until the `paraML` runtime system is invoked. The runtime system is called by applying the `paraml` function to a thunk expression, which is the top-level user expression. This expression in turn may invoke processes. The type of the `paraml` function is:

```
val paraml : (unit -> 'a) -> unit
```

The outline of an appropriate structure that invokes the first function definition for `factorial` is as follows:

6. It is expected that new operations will be provided that permit group and process ids to be retrieved given a process name, and will be called `group_number_from_name` and `process_number_from_name`.

```

structure Main =
  struct
    open ParaML

    fun factorial (n:int) = (create (no_ports())
      (fn () =>
        if n = 0
        then 1
        else
          let val execproc = factorial (n-1)
              val part_fac = result execproc
          in n * part_fac
          end
        end))

    fun doit () = let val fact10 = result (factorial 10)
                  in (print "Factorial of 10 was: "; print fact10; print
"\n")
                  end

    val _ = paraml doit
  end

```

In this structure, we firstly open the `ParaML` structure which defines the various `paraML` operations and also contains the `paraML` runtime system (written in ML of course) which executes on the cells. The `factorial` process definition is defined, and a function called `doit`, which calls `factorial` with the argument 10, obtaining its result and printing it out. The `doit` function will be the top-level user expression. Lastly, we use a side-effecting `val` declaration to call the `paraml` function with `doit`, and the processes will be created, execute, return their results, the value 3628800 will be printed out, and the program will terminate. Details on how to compile and run an actual `paraML` program are given in Section 4.

3.5 Concurrent ML

Two functions are provided to support the integration of Concurrent ML modules with `paraML`. These functions are `setCurThread` and `getCurThread`, which in Reppy's original versions used the `System.Unsafe.setvar` and `System.Unsafe.getvar` routines. These are used by the `paraML` runtime system however to maintain process state information on a per-cell basis, so alternative implementations have been provided. The types of these operations are:

```

val setCurThread : 'a -> unit
val getCurThread : unit -> 'a

```

3.6 Miscellaneous

3.6.1 Printing

There are a small number of miscellaneous functions which are also of use in `paraML` programs, but which aren't a strong part of the language design. The first operations are `p_print` and `p_flush`, which access a per-process output buffer. Currently, all information produced from

an execution of a paraML program is only accessible from the standard output mechanisms.⁷ Since standard output must travel on the AP1000's B-net, and is therefore heavily dependent on processing load on the AP1000 host machine, these operations have been provided which store up string output in a single buffer with the `p_print` operation, and the buffer is flushed either explicitly with the `p_flush` operation or implicitly on process termination. The types of these operations are:

```
val p_print : string -> unit
val p_flush : unit -> unit
```

3.6.2 Configuration Information

In order to allow the user to write programs that are portable across different AP1000 configurations, the operation `nodes_for_processes` is provided which returns an integer value of the number of cells which will be used for executing processes on. The number of processes can exceed the number of nodes, but some programs may want to know the exact number of processes to create so that there is only one process per cell, or to avoid `groupcreate` errors. The number of nodes available is currently the number of cells configured minus two (the first is used to manage allocation of processes to cells, and the second is used to execute the top-level user expression). The type of the `nodes_for_processes` operation is:

```
val nodes_for_processes : unit -> int
```

3.6.3 Environment Information

Execution of programs with parameters dependent on an environment variable declared on the host machine is supported through the provision of the `getenv` operation. This operation takes a single string argument which is the name of the environment variable to lookup, and returns the string value that matches that name. The implementation of this operation requires cell-host and host-cell communications, so is relatively expensive, and should be kept outside of timing analysis. The type declaration of the `getenv` operation is:

```
val getenv : string -> string
```

3.6.4 Process Scheduling

Lastly, since there is no pre-emptive process scheduling with paraML, access to the descheduling primitive `yield` is provided. This operation simply deschedules the executing process, which places it into a queue of processes ready to run again on being rescheduled by the paraML runtime system. The type declaration of the operation is:

```
val yield : unit -> unit
```

4. Compiling and Running

There are a number of different components of the paraML system, which reside beneath the top level directory `paraml`.

1. the modified `sml2c` compiler (in `bin` and `src/paraml`)

7. In future, the SML/NJ stream I/O mechanisms will be integrated with the Acacia parallel file system to provide file input and output facilities. An interactive version of the system may also become available.

2. ML runtime system libraries compiled for the host, cell, and normal SML2C execution (in `lib` and `src/paraml/[host|cell]runtime`)
3. the `host` program, which executes a program called `cell` (in `System/Host`)
4. the `paraML` runtime system which executes on every cell (in `System/Cell`)
5. the user program, whose final executable image is called `cell` (examples of user programs are in subdirectories of `examples`)
6. there is also the system which incorporates Concurrent ML primitives (in `CML`)

To compile these components the following should be performed:

1. Users should not attempt to alter the `sml2c` compiler unless they are very sure of what they are doing.
2. The ML runtime libraries can be recompiled with a `makefile` provided (in the current release directory hierarchy) in `paraml/src/paraml`. To completely remake the libraries, the commands `make clean` followed by `make` should be executed.
3. Users should not attempt to change the files in the `System/Host` directory unless they are sure of what they are doing. The program can be recompiled by executing the commands `make clean` followed by `make` in the directory.
4. Users should not attempt to change the files in the `System/Cell` directory unless they are sure of what they are doing. The `paraML` runtime system can be recompiled by executing the commands `make clean` followed by `make` in the directory.
5. The user programs in the `examples` directory can be recompiled by executing the commands `make clean` followed by `make` in the respective directories. Users developing their own `paraML` programs should create new directories with `makefiles` created like the ones given in the `Example` subdirectories.
6. Users wishing to utilise the `CML` primitives should be able to do this by following the same approach as the example program in the directory `CML`.

A `paraML` program can be executed by executing the command `./host -WF out`, where `out` is some output filename, in the directory in which their `cell` program has been compiled. The messages “Starting program” and “Done” indicate successful completion of the program.

Appendix A – Signature File for the ParaML Structure

```
signature PARAML =
  sig
    type 'la PortName
    type 'la Ports
    type ('r,'c) ProcessName

    exception NoPorts
    exception PortNotOwned
    exception BadGroupCreate
    exception ParaMLExit

    val p_print : string -> unit
    val p_flush : unit -> unit

    val getenv : string -> string

    val selfid : unit -> ('r,'lc) ProcessName

    val process_number : unit -> int
    val group_number : unit -> int
    val nodes_for_processes : unit -> int

    val port : unit -> 'la PortName
    val ports : 'la -> 'la Ports
    val noports : unit -> unit Ports

    val getports : ('r,'lc) ProcessName -> 'lc
    val selfports : unit -> 'lc

    val send : 'la PortName -> 'la -> unit
    val broad : 'la PortName -> 'la -> unit
    val get : 'la PortName -> 'la

    val ready : 'la PortName -> (unit -> bool)
    val select : ((unit->bool) * (unit->unit)) list -> unit

    val create : 'lc Ports -> (unit -> 'r) -> ('r,'lc) ProcessName
    val groupcreate : 'lc Ports -> (unit -> 'r) -> int ->
      ('r,'lc) ProcessName list
    val result : ('r,'lc) ProcessName -> 'r

    val paraml : (unit -> 'a) -> unit

    val yield : unit -> unit
    val setCurThread : 'a -> unit
    val getCurThread : unit -> 'a
  end
end
```

References

- [1] AT&T Bell Labs, Standard ML of New Jersey version 0.93 Documentation, (1993).
- [2] Andrew Appel and David MacQueen “A Standard ML Compiler” in *Functional Programming Languages and Computer Architecture* pp. 301–324. Springer–Verlag, 1987. Vol. 274 Lecture Notes in Computer Science.
- [3] Peter Bailey, “paraML: a parallel extension of ML,” B.Sc.(Hons) Thesis, Dept. Comp. Sci, Australian National University, (1991).
- [4] Peter Bailey and Malcolm Newey, “The Implementation of ML for the Fujitsu AP1000,” published in *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, (June 1992).
- [5] Peter Bailey and Malcolm Newey, “The Implementation of ML for Distributed Memory Multiprocessors,” to be published in *SIGPLAN Newsletter – dedicated issue on the proceedings of the Boulder Workshop on Languages, Compilers, and Run–Time Environments for Distributed Memory Multiprocessors*, (October 1992).
- [6] Peter Bailey & Malcolm Newey, “An Extension of ML for Distributed Memory Multicomputers,” in *Proceedings of the Sixteenth Australian Computer Science Conference*, ed. Gopal Gupta, George Mohay, Rodney Topor, vol. 15, 1, pp. 387–396, (February 1993).
- [7] Robin Milner, Mads Tofte, and Robert Harper, *The Definition of Standard ML*, MIT Press, (1990).
- [8] Robin Milner and Mads Tofte, *Commentary on Standard ML*, MIT Press, (1990).
- [9] Malcolm Newey, “Towards a CAP Implementation of ML,” *Proceedings of the First CAP Workshop*, Kawasaki, Japan, (November 1990).
- [10] Larry Paulson, *ML for the Working Programmer*, Cambridge Press, (1992).
- [11] John Reppy, “Higher–Order Concurrency,” Ph.D Thesis, Dept. Comp. Sci., Cornell University, (June 1992).
- [12] John Reppy, “CML,” in *Proceedings of Programming Language Design and Implementation*, 1992.
- [13] David Tarditi, Anurag Acharya, and Peter Lee “No Assembly Required: Compiling Standard ML to C.” Technical Report 187, CMU–CS, (November 1990).
- [14] Jeffrey D. Ullman, *Elements of ML Programming*, Prentice-Hall, (1993).