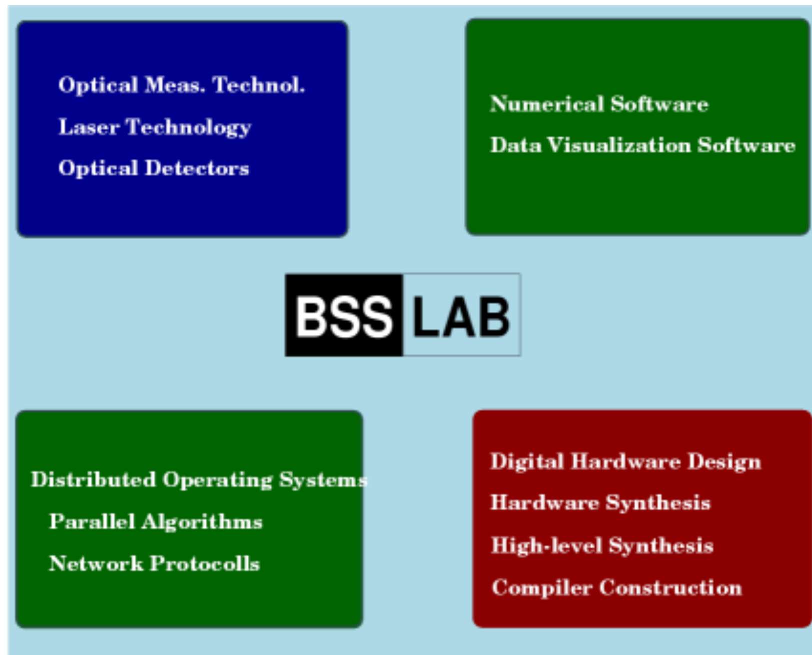
WWW.BSSLAB.DE



Independent Research and Development Laboratory

Scientific Measuring and System Techniques

BSSLAB – Dr. Stefan Bosse
Vohnen Street 86
D–28201 BREMEN

Germany

Phone: (49)421/21864103 or (49)421/875215
Fax: (49)421/875215

VAM – the functional approach

The short name VAM is the abbreviation for the **Virtual Amoeba Machine**. The concepts of a native Amoeba system running with its own kernel and the AMUNIX Amoeba emulation as an addon for UNIX operating systems have many advantages. But there are some important disadvantages of these traditional concepts:

1. All parts of the kernel, the library and user space programs are programmed in the C language. C is a powerfull lowlevel language, but it lacks of safety and the ability of abstraction. There are too many risks of failures, mostly related with the always visible pointer handling. Moreover, the programmers spent a lot of time and power with resource management, like memory space for data structures.
2. C programs are compiled and linked for the native microprocessor code only. This is no problem for the (of course portable) VX–Kernel, currently only supporting the i386 architecture. But the AMUNIX emulation should run on various different operating systems and microprocessors. For n different operating systems and m hardware architectures, youn need n*m builds of the AMUNIX system, all the libraries and util programs, the flip protocol stack and so on.

Some disadvantages can be eliminated with new and modern concepts of **functional programming** instead using memory pointer based languages like C with a high probability to cause unresolved errors in the program code during program execution somewhere in time and space. The authors experiences showed in the past, that programming errors due to wrong pointer handling can occur years (!!!) after the program was written. These class of programming errors are really hard to find out, because wrong pointer handling causes normally no exception directly. Instead, some part of the program memory will be corrupted. This will cause a fully undefined program behaviour with program aborts in parts of the program not related to the original pointer corruption. More than 90% of the C code in the world is infected by pointer corruption and executes in an undefined way.

Functional programming, especial the **ML programming language**, features a strong typed data system which avoids several common programming errors, like integers of different binary widths which can cause unpredictable program output. Moreover, ML provides the programmer with an automatic resource management. There is no memory pointer code needed (and of course allowed) to program an algorithm. True functional code has a predictable runtime behaviour.

In contrast to the machine C language the ML languages provides the programmer with some kind of abstraction facilities. Functional programming is a style that uses the defintion and application of functions as essential concepts *[COU98]* . In this approach, variables play the same role as in mathematics: they are the symbolic representation of values and, in particular, serve as the parameters of functions.

In imperative programming languages like C there are some parasites: each function must be introduced with an arbitrary name F. In those languages, we cannot define a function without naming it and there is an explicit assignment operation, like the return operation. Using imperative languages, the user must provide the compiler with the type of functions or variables. In contrast, in functional programming the compiler is responsible to find out types of functions and variables (or data structures)! **Types exist in ML, too, though they are computed by the system.**

A functional style tends to preserve the simplicity and spirit of mathematical notation. Because functions can be denoted by expressions, function values can be treated as values just like all others and therefore functional values can be passed as argumentes to and returned by other functions.

For reasons explained below, the **OCaML programming language** from INRIA software institute was choosen for implementing the VAM system. OCaML is a kind of ML language, but not fully compatible to standard ML. It provides a ML core with a powerfull and easy to use module system. Additionally, it has an object orientated class system built on the top of the ML core.

To illustrate the power of functional programming, lets make an example:

```
fun x -> 2 * x + 1
```

This is simply an expression for a mathematical function. Using C, we need to write instead:

```
int F(int x){
    return (2*x+1);
}
```

A named function value, for example "f", can be expressed with the *let* operator:

```
let f x = 2 * x + 1
```

The ML compiler will compile this expression and a result is the following derived type interface:

```
# val f : int -> int
```

You can see, that there is no necessatiy of a type declaration if you define a (functional) value. The compiler will evaluate the expression and finds out that the multiplication and addition operator is from type integer (int), and therefore the function argument and the result of the function must be from type integer, too.

The fact that the ML language treats functions as generic functional values like normal "variable" values can clearly shown by the next two examples:

```
let f x = x + 1
# val f : int -> int
let x = 2
# val x : int
```

Here, the first line defines a function expecting one argument, and the second definition just defines a symbolic variable (not mutable) from type integer. This value just returns a constant.

Another powerfull of ML is **polymorphism**. That means, the compiler can't evaluate one or more types of functional values inclusive the return value of a fucntion. This leads to a powerfull and mighty instrument for reuseable code. For example a function which iterates a list entry by entry and passes each list entry to a user supplied function which make some nice things with this list entry:

```
let rec list_iter func list =
    match list with
    | hd::tl -> func hd; list_iter func tl;
    | [] -> ();
;;

# val list_iter : ('a -> 'b) -> 'a list -> unit = <fun>
```

The interface derived by the compiler specifies no concrete type. The only fact we know is that the first argument must be a function wich expects as the first argument a list entry from same type as specified in the second list argument `'a list`. The native lists supported by ML are simple single linked linear lists. Lists can hold data of arbitrary types. The :: operator in the match statemant, compareable with the switch/case statemant in C, splits the list into a single value, the head of the list, and a remaining tail list. The [] operator is the empty list. The last example showed one more feature of the ML language: **function recursion**.

Another kind of data type leads to a more strcutured programming style: tuples. These are simply spoken unnamed compounds of arbitrary data types and entries. Lets assume you want to return more than one value from a function. In C you must use several pointers passed to the function with its arguments. But clearly, function arguments should of type input, not of type output. Using ML, there is a solution, called data tuples.

The following example shows this powerfull feature, with a function expecting two arguments of type integer, and returns a tuple of three integers:

```
let arithm x y =
    let mul = x * y in
    let div = x / y in
    let add = x + y in
    (mul,div,add)
;;
# val arithm : int -> int -> int * int * int = <fun>
# arithm 1 2
# - : int * int * int = 2, 0, 3
```

A final example shows the usage of the above defined polymorph function *list_iter*:

```
let list = [1;2;3] ;;
# val list : int list = [1; 2; 3]

let sum = ref 0 ;;
# val sum : int ref = {contents = 0}

list_iter (fun x -> sum := !sum + x) list ;;
print_int !sum
# 6
```

The first line defines a list with three entries from type integer. The second one defines a traditional variable known from imperative programming languages. This imperative variable is mutable, as shown in the nameless function in the *list_iter* function evaluation call. The "!" operator just returns the current value of this variable, and the ":=" operator assign a new value to the variable. But in fact this is not a traditional variable, it's a reference to an object. Each time a new value is assigned to this reference, this reference points to a new object! In the above example it's just the result of the addition of two values – a constant in this case.

But back to the motivations for using functional programming for the extensive job of operating system programming. As shown above in various examples, the **programmer spent his time with implementing an algorithm**, and not with resource allocation and release, like in imperative languages like C. This make especial rapid prototyping more faster and safe. This can lead **to a more clean and structured programming** style. In CaML there are references, too. But they must point always to valid objects. There is no nil pointer like in C. And therefore memory access violations due to nil pointers are not possible. This reduces the time spent in the job of programming of about thousand of hours, really belive it.

But that's not all, folks. The OCaML language has one more powerfull feature: the ML compiler doesn't produce native assembler code directly executed by the host machine, no, it produces architecture and machine independent code, called bytecode. This bytecode is then interpreted by a **virtual machine**, emulating an abstract and in the case of OCaML nearly perfect and to the ML language highly adpated

execution machine. This virtual machine hides all system dependencies from the underlying host operating system. This feature is perfectly suited for the implementation of a **portable operating system environment** !

The OCaML virtual machine is a traditional stack based bytecode processor with memory allocation and delayed freeing of no more needed memory by a background garbage collector. Experiencences showd that an algorithm executing with OCaML using the virtual machine approach is only about 4–5 times slower in execution time than an optimized C program. The required memory space is hard to predict, perhaps in contrast to C programs. It can be of course higher than by a compareable C program. So, for embedded microcontrollers with hard resource constraints, the C language is mostly the better choice.

Now, with the functional approach in mind, it seems to be a simple task to implement distributed operating system concepts using the OCaML VM and the ML language. Indeed, the original OCaML virtual machine is highly portable. The virtual machine consists of these main parts:

1. The bytecode interpreter with a stack based CISC machine architecture. It's mainly one C function unrolling all the bytecode instructions (about 140 instructions),
2. several hardcoded ML standard function groups: Arrays, Lists, Strings, Integers, Floats...,
3. support for custom datastructures not interpreted by the VM (handled in external fucntions),
4. the memory manager and the garbage collector,
5. some system dependent parts (the Unix and System module),
6. IO handling (terminal and file input & output),
7. backtracing and debugger support.

One result of the functional approach of ML is that functional values can be evaluated independently. This offers a great advantage for interactive toplevel systems. OCaML is equipped with an interactive interpreter system. You can either type instructions on the input line, or read input from a file. This text input is compiled (evaluated) to bytecode and can be immediately executed. These on the fly compiled code executes with the same runtime behaviour than traditional bytecode externally compiled directly using the compiler.

The OCaML system, both the virtual machine and the runtime system (VM) , was adapted to the demands of the Amoeba operating system concepts. The VM was improved, and the bytecode compiler gots some enhancements.

One main feature of the OCaML virtual machine is a simple interface of user customized C functions accessible from ML code. For example a function allocating a new Amoeba port is implemented in an external C function:

```
CAMLexport value ext_port_new (value unit)
{
    CAMLparam0();
    CAMLlocal1(port_v);
    CAMLlocal1(port_s);
    port_v = alloc_tuple(1);
    port_s = alloc_string(PORTSIZE);
    memset(String_val(port_s),0,PORTSIZE);
```

```
    Store_field(port_v,0,port_s);
    CAMLreturn(port_v);
}
```

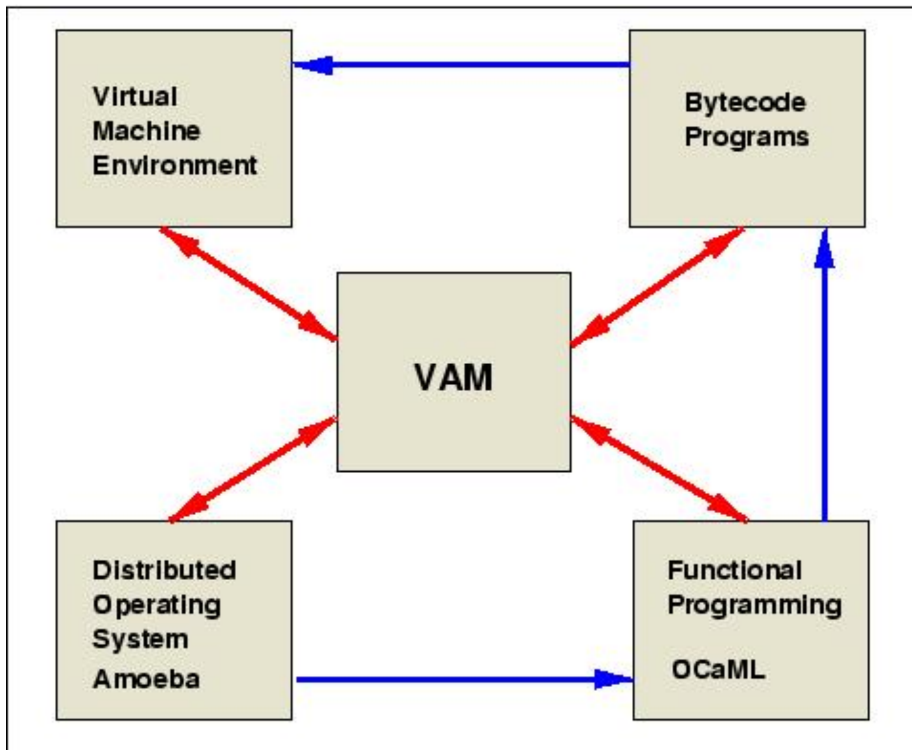
The ML code, which tells the compiler that the function is external, is now very simple:

```
type port = Port of string (* length = PORTSIZE *)
external port_new: unit -> port = "ext_port_new"
```

Each time the *port_new* ML function is called, the virtual machine will call the *ext_port_new* C function. The virtual machine is implemented only with a library and a main source code file created dynamically. Therefore, the virtual machine can be recompiled any time with additional functionality. This feature was an important starting point for VAM.

Most of the Amoeba modules known from the C world were reimplemented with ML. Only a small part is linked inside the virtual machine, namely the AMUNIX interface for threads and RPC communication. All other parts are created from pure ML source.

The basic concepts of the VAM system are shown in figure1.



(Fig. 1) The basic conecpts of the virtual amoeba machine.

The **VAM system** is divided into a development and a runtime environment. The development envrionmnet provides the following parts:

1. Several **ML libraries** with different modules implementing the **ML core** concepts, the interface to the **UNIX** environment, the **Amoeba** system, building the largest part, and a **graphical widget** library.
2. A standalone **ML Bytecode compiler** producing bytecode exectubales. Additionally, this compiler can compile a new custom designed virtual machine. The bytecode compiler is completely programmed in ML, too.
3. An **interactive toplevel VAM** program. This program, simply called *vam*, contains the bytecode compiler, a command line like toplevel shell for user interaction, and all ML Modules. With this interactive system it's possible to compile expressions directly entered into the command line or load external ML scripts, compiled on the fly, too. This on the fly compiled bytecode is linked to the current bytecode program during runtime and can be executed like any other built in function.

The following list (in alphabetic order) gives an overview of currently implemented VAM modules. Some modules were provided by external programmers. They were modified and adapted to VAM.

- **Afs_client** – Client interface to the Amoeba Filesystem Server (AfS).
- **Afs_cache** – a data cache implementation used by both the AFS and DNS server.
- **Afs_common** – types and structures common to server and client.
- **Afs_server** – core of the AFS server.
- **Afs_server_rpc** – the RPC server loop.
- **Amoeba** – the Amoeba core library implementing basic concepts like capabilities.
- **Ar** – ASCII representation of Amoeba structures like capabilities.
- **Arg** – Parsing of command line arguments. *[OCAML305]*
- **Array** – Array operations. *[OCAML305]*
- **Bootdir** – support for Amoebas bootdirectory system.
- **Bootdisk_common** – Kernel Boot directory and disk module server implementation. This server module emulates a DNS/AFS interface for the kernel boot partition holding kernels, binaries needed for bootstrap purposes and configuation files with a very simple filesystem.
- **Bootdisk_server** – the core module of the server.
- **Bootdisk_server_rpc** – the RPC server loop.
- **Bstream** – Generic Bytestream interface
- **Buf** – Provides machine independent storing and extracting of Amoeba structures in and from buffers with bound checking.
- **Buffer** – Extensible buffers. *[OCAML305]*
- **Bytebuf** – Low level Buffer management. Used for example by the Rpc module.
- **Cache** – Provides a fixed table cache.
- **Callback** – Registering Caml values with the C runtime for later callbacks. *[OCAML305]*
- **Cap_env** – support for Amoebas capability environment, similar to UNIX string environment.
- **Capset** – capability sets
- **Circbuf** – Support for circular Amoeba buffers with builtin synchronization primitves needed in a multithreaded program environment.
- **Char** – Character operations. *[OCAML305]*
- **Db** – debug support.
- **Dblist** – double linked lists.

- **Des48** – Crypthograpic de– and encoding.
- **Digest** – Message digest (MD5). *[OCAML305]*
- **Dir** – High level directory service interface.
- **Disk_client** – Virtual Disk Server client interface which provides unique low level access to logical (partitions) and physical disks.
- **Disk_common** – common part used by server and client.
- **Disk_pc86** – i386 dependent parts.
- **Disk_server** – the disk server, running outside the kernel (UNIX).
- **Disk_server_rpc** – the server loop.
- **Dns_client** – Directory and Name service (DNS) client interface
- **Dns_common** – types and structures of the DNS common for client and server modules.
- **Dns_server** – the core module of the Directory and Name server DNS.
- **Dns_server_rpc** – the server loop.
- **Filename** – Filename handling. *[OCAML305]*
- **Format** – Pretty printing. *[OCAML305]*
- **Gc** – Memory management control and statistics; finalised values. *[OCAML305]*
- **Genlex** – A generic lexical analyzer. *[OCAML305]*
- **Hashtbl** – Hash tables and hash functions. *[OCAML305]*
- **Imagerpc** – Image transfer utils.
- **Int32** – 32–bit integers. *[OCAML305]*
- **Int64** – 64–bit integers. *[OCAML305]*
- **Ksys** – Kernel system client interface.
- **Ktrace** – Kernel trace and debug support.
- **Layz** – Deferred computations. *[OCAML305]*
- **Lexing** – The run–time library for lexers generated by [ocamllex]. *[OCAML305]*
- **List** – Single linked List operations. *[OCAML305]*
- **Machtype** – Machine type representation, similar to OCaML's int32 and int64 module, but more general. Remember that OCaML integer are only 31/63 bit wide! The last bit is used internally. So, when the bit length must be guarenteed, use THIS module.
- **Map** – Association tables over ordered types. *[OCAML305]*
- **Marshal** – Marshaling of data structures. *[OCAML305]*
- **Monitor** – Server event monitor support.
- **Mutex** – Supports Mutual Exclusion locks.
- **Name** – Amoeba name interface (easy to handle frontend to DNS) support.
- **Om** – the core module of the Object Manager Server (Garbage Collector).
- **Parsing** – The run–time library for parsers generated by [ocamlyacc]. *[OCAML305]*
- **Pervasives** – This module provides the built–in types (numbers, booleans, strings, exceptions, references, lists, arrays, input–output channels, ...) and the basic operations over these types. *[OCAML305]*
- **Printf** – Formatted output functions. *[OCAML305]*
- **Proc** – Amoeba process client interface. Provides functions to execute (native) Amoeba binaries.
- **Queue** – First–in first–out queues. *[OCAML305]*
- **Random** – Pseudo–random number generator (PRNG). *[OCAML305]*
- **Rpc** – the fundamental communication interface.
- **Sema** – Semaphore synchronization support.
- **Set** – Sets over ordered types. *[OCAML305]*
- **Shell** – some utils for shell like programs.
- **Signals** – Adaption of Amoeba signals to VAM.
- **Sort** – Sorting and merging lists. *[OCAML305]*
- **Stack** – Last–in first–out stacks. *[OCAML305]*
- **Stdcom** – this module implements most of the Amoeba standard commands like *std_info* .

- **Stdcom2** – some more.
- **Stderr** – defines Amoeba standard errors.
- **Stream** – Streams and parsers. *[OCAML305]*
- **String** – String operations. *[OCAML305]*
- **Sys** – System independent system interface... *[OCAML305]*
- **Thread** – Amoeba multithreading module.
- **Unix** – interface to the UNIX operating system (similar to C functions).
- **Vamboot** – a core module implementing Amoeba boot services.
- **Virtcirc** – virtual circuits: distributed access of circular buffers.
- **Weak** – Arrays of weak pointers. *[OCAML305]*
- **WX_adjust** – X widget library: Adjustement object. *[Fab99]*
- **WX_bar** – X widget library: Horizontal and vertical basic widget container. *[Fab99]*
- **WX_base** – X widget library: Base object. *[Fab99]*
- **WX_button** – X widget library: Button object. *[Fab99]*
- **WX_dialog** – X widget library: Dialog object. *[Fab99]*
- **WX_display** – X widget library: X display interface. *[Fab99]*
- **WX_filesel** – X widget library: Fileselection menu object. *[Fab99]*
- **WX_image** – X widget library: Generic image widget. Derived from the WX_pixmap class. *[Fab99]*
- **WX_label** – X widget library: Text label object. *[Fab99]*
- **WX_ledit** – X widget library: Text input object. *[Fab99]*
- **WX_object** – X widget library: Basic WX object support. *[Fab99]*
- **WX_popup** – X widget library: Simple popup menu object. *[Fab99]*
- **WX_progbar** – X widget library: Progress/Value bar object.
- **WX_radiobutton** – X widget library: Radio button object. *[Fab99]*
- **WX_root** – X widget library: Parent object for all toplevel windows. *[Fab99]*
- **WX_screen** – X widget library: X interface utils. *[Fab99]*
- **WX_slider** – X widget library: Slider object.
- **WX_table** – X widget library: Table container for WX objects. *[Fab99]*
- **WX_tree** – X widget library: Tree selector object. *[Fab99]*
- **WX_types** – X widget library: Basic types. *[Fab99]*
- **WX_valbar** – X widget library: Value bar object.
- **WX_viewport** – X widget library: Encapsulates WX objects. *[Fab99]*
- **WX_wmtop** – X widget library: X window manager interface. *[Fab99]*
- **X** – the core X11 graphic windows module. Enables direct X11 programming under VAM. Both UNIX sockets and Amoebs Virtual Circuit X11 communication is implemented. *[Fab99]*
- **XGraphics** – machine–independent graphics primitives.
- **Ximage** – Generic image support. *[Fab99]*
- **Xtypes** – basic X11 types and structures. *[Fab99]*

With this incredible amount of VAM and OCaML modules (and there are still many more) several Amoeba servers, administration and util programs are implemented to built a fully functional distributed operating system either on the top of UNIX or a more raw version on the top of the VX–kernel.

VAM Amoeba servers:

- **AFS**: the atomic filesystem server with backends for UNIX (the filesystem is stored in generic UNIX files or harddisk partitions managed by UNIX) and Virtual Disks (the filesystem is stored on harddisks managed by the VX–Kernel). The AFS filesystem consists of an inode partition holding

filesystem informations about each file (described by one inode) and the data partition(s) holding the file data indexed by the file inode.

- **DNS**: the directory and name server. There are slightly different versions for UNIX and VX−Amoeba. The DNS system consists of an inode partition which holds directory capabilities and some basic. The directories itself are saved in generic AFS file obecjts.
- **VAMBOOT**: boot services for an initial operating sysetm startup. The boot server is in fact only a ML script using the boot module.
- **VOM**: a garbage collector server. This server is responsible to cleanup servers periodically. For example the fileserver can contain file object not referenced anymore, directly speaking the capability of the file object is lost. The OM server is a ML script, too.
- **VDISK**: a virtual disk server providing a virtual disk interface for UNIX devices.
- **BDISK**: a bootdirectory server using the virtual disk interface either of native Amoeba or local UNIX devices.

VAM administration and util programs:

- **vash**: the VAM shell. It's a user interactive command line controlled shell compareable with UNIX bash, providing most of the Amoeba administrative and standard commands.
- **xafs**: a graphical frontend both to the Amoeba directory and filesystem and the UNIX filesystem allowing easy data transfer between both worlds.
- **std**: Amoeba standard commands directly accessible from the UNIX shell.
- **vax**: Executes native Amoeba binaries either located in the Amoeba AFS/DNS system or in the local UNIX filessystem on a specified native Amoeba host. Amoeba kernels can be rebooted with this tool, too.

References

[KAS93]
M.F. Kaashoek, R. van Renesse, H. van Staveren, and A.S. Tanenbaum
FLIP: an Internetwork Protocol for Supporting Distributed Systems
ACM Transactions on Computer Systems, pp. 73−106, Feb. 1993.

[AMSYS]
Amoeba 5.3 System Administration Guide

[AMPRO]
Amoeba 5.3 Programming Guide

[COU98]
G. Cousineau, M. Mauny
`The Functional Approach to Programming`
Cambridge Press, 1998

[Fab99]
Software: Fabrice Le Fessant, projet Para/SOR, INRIA Rocquencourt.
[OCAML305]
Software: OCaML version 3.05, Xavier Leroy et al., projet Cristal, INRIA Rocquencourt

---

Generated by MANDoc (C) 2005 BSSLAB Dr. Stefan Bosse
Revision 1113144376

(C) BSSLAB Dr. Stefan Bosse, Revision 1138111946