Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Technical
Memo
TM-98-07

The RAWAM: Relfun-Adapted WAM Emulation in C

Markus Perling

December 1998

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210
E-Mail: info@dfki.uni-kl.de

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: + 49 (681) 302-5252
Fax: + 49 (681) 302-5341
E-Mail: info@dfki.de

WWW: http://www.dfki.de

The RAWAM: Relfun-Adapted WAM Emulation in C

Markus Perling

December 21, 1998

Abstract

This work describes the C implementation of the Relfun-Adapted WAM (RAWAM). The RAWAM is an abstract machine tailored to the relational-functional language Relfun, designed and implemented on the basis of the Warren Abstract Machine (WAM). Its goal is to replace an older LISP-implemented Relfun WAM by delivering comparable functionality at higher speed. The RAWAM implementation is introduced by reference to Hassan Aït-Kaci's book "Warren's Abstract Machine: A Tutorial Reconstruction", and the present work will emphasize the differences and extensions w.r.t. this book. These include an assembler, an optimizer, a rudimentary module system, a more flexible realization of the standard WAM memory layout, as well as Relfun-specific extensions for functional and relational builtins, sorts, generalised indexing, and a simple higher-order facility. The implementation of the RAWAM will be described in terms of pseudo code and schematic patterns for the data structures. A relational-functional benchmark revealed a speed-up factor of 20-30 of the RAWAM compared to the older WAM.

Contents

1 Introduction 4

2 Notions 5

3 Overview 6
- 3.1 File Interface 7
- 3.2 What RAWAM gets from the RFM Compiler 7
  - 3.2.1 Sort Data 8
  - 3.2.2 WAM Routines 8
- 3.3 Supported RelFun Language and Compiler Features 9
  - 3.3.1 Basic Instruction Set and Data Types 9
  - 3.3.2 Generalized Indexing 9
  - 3.3.3 Cut 9
  - 3.3.4 Higher Order 10
  - 3.3.5 Arithmetics 10
  - 3.3.6 Types 10
  - 3.3.7 Extra Instructions 10

4 Global Organisation 10

5 Modules 12
- 5.1 Using Modules 12
- 5.2 Semantics of Modules 12
- 5.3 Standard Modules 13

6 Hash Tables 13

7 The Byte Code 14

8 Generation of the Internal Program Representation 15
- 8.1 Tokenizer 15
- 8.2 Assembler 17
  - 8.2.1 `getsortbase` 17
  - 8.2.2 `getcode` and `getquery` 18
- 8.3 The Optimizer 19

9 WAM Registers 19

10 WAM Extensions 20
- 10.1 WAM Data Types 20
- 10.2 Sorts 20
- 10.3 Higher Order 23

11 Heap, Stack, Trail, PDL 24
- 11.1 General Handling of HST 25
- 11.2 Heap 26
  - 11.2.1 Data 26
  - 11.2.2 Access 26
- 11.3 Trail 30
  - 11.3.1 Trailcells 30
  - 11.3.2 Access 30
- 11.4 Stack 32
  - 11.4.1 Layout 33
  - 11.4.2 Stackcells 33
  - 11.4.3 Timestamps 34
  - 11.4.4 Access 34

12 Auxiliary Items **35**
12.1 Memory Management for Linked Lists 35

13 Conclusions **36**

A Some Corrections **38**

B Overview: Supported WAM Instructions **38**
B.1 Instructions as in Aït-Kaci's description of the WAM 38
B.1.1 `Put` instructions 38
B.1.2 `Get` instructions 38
B.1.3 `Unify` instructions 38
B.1.4 `Control` instructions 39
B.1.5 `Choice` instructions 39
B.1.6 `Indexing` instructions 39
B.1.7 `Cut` instructions 39
B.2 Additional Instructions 40
B.2.1 Arithmetics Instructions 40
B.2.2 Miscellaneous Instructions 41
B.2.3 Integrating Instructions 41

C A Demo Script **42**

1 Introduction

In this work we describe the C implementation of the Relfun-Adapted WAM (RAWAM). This implementation arose out of the author's attempt to understand the principles of the Warren's Abstract Machine ([War83], [War77], [GC96], [AK91], [GLLO85], [VR94], [MW88], [Rus92]). It came along with the question of how to increase the speed of the LISP-implemented abstract machines for our relational-functional programming language, RelFun ([Bol97]).
The RAWAM is now the first component of the RelFun implementation which is fully realized in a low-level language, and it succeeded in both, providing a clarified understanding of the WAM and an increased execution speed for RelFun programs, currently by a factor of 20 to 30 compared to the fastest earlier implementations (the absolute speed of the RAWAM reaches approximately 350 KLIPS on an UltraSPARC hardware). However, RelFun's compiler was not designed to be high-level optimizing and also requires the RAWAM to support more extended features such as sorts. Moreover, the RAWAM's implementation is kept as portable as possible; so, e.g., no machine-dependent assumptions on the size of data structures were made. Further, the transition to an even lower-level native-code compilation has not yet been done; but for this the RAWAM can serve as an intermediate step. A good demonstration of RelFun's enhanced speed and, incidentally, an important testbed for the RAWAM, is the implementation of the GeneTS genetic algorithm using methods of declarative programming in [Per97]. The GeneTS work couldn't have been done without such a fast enough execution model.
RelFun utilizes several execution principles - besides the interpreter, there is a WAM, called GWAM, which originated from the work of Nystrøm ([Nys85]), and a functional abstract machine, LLAMA, both described in [Sin95] and [BEH$^+$96]. RelFun's compilation logic mostly relies on the WAM part, so it was a natural choice to advance to a C implementation of the RelFun WAM. Indeed, the RAWAM's goal is to be a full replacement of the GWAM.
The RAWAM system includes an assembler, a small optimizer, a rudimentary module system, a more flexible realization of the standard WAM memory layout, as well as many Relfun-specific extensions, an overview of which is given in subsection 3.3.
The description that follows will base on Hassan Aït-Kaci's book "Warren's Abstract Machine: A Tutorial Reconstruction" ([AK91]) (while this book is out of print, an online postscript version for non-commercial use can be found at [AK95]). All features that hallmark the RAWAM are extensions to the WAM described therein. Along these lines, we will emphasize mainly the differences and extensions w.r.t. to this book.
Because the RAWAM is so related to the WAM, we will try to describe its implementation in such a way that it can be easily enhanced and adapted to other logic programming systems. The description will be terms of abstract patterns for the data structures and pseudo code for parts of the RAWAM which are essential for describing the extensions w.r.t. the original WAM design. The source can be found at [Per].
The contents of this work will be as follows: Section 2 explains the notions used in the subsequent sections. Section 3 will give an overview for the whole RelFun system, how the RAWAM fits into it, and which RelFun features the RAWAM supports. Section 4 will give an overview of the structure of the RAWAM. Sections 5 to 7 describe structures relevant for all parts of the RAWAM. Section 8 describes the assembler for the RAWAM. Sections 9 to 11 reveal how the abstract machine is built. In appendix A we give a reference to some corrections to [AK91] and appendix B gives an overview of all implemented RAWAM instructions. Appendix C, finally, contains an actual session script that demonstrates some of the RAWAM's facilities and the general speedup compared to the GWAM.

2 Notions

In the next sections data structures used in the implementation of the RAWAM will be described. These are mainly C-structures or C-unions, often used for linked lists. The contents of a structure or union are schematically presented in boxes containing type and aim of each entry. E.g. structures are represented as:

name of structure		
name	type	purpose
name	type	purpose
.		
.		
.		

A union is similarly represented, but with a thick rim at the left side and the name of the union omitted:

name	type	purpose
name	type	purpose
.		
.		
.		

If a linked list is built from a structure, this is indicated by one or two arrows, when the list is singly or doubly linked, respectively:

$\longrightarrow$		
name	type	purpose
.		
.		
.		

$\longleftarrow$ $\longrightarrow$		
name	type	purpose
.		
.		
.		

To give a concrete example, consider the data scheme:

Indiv_entry		
$\longrightarrow$		
ctag	tag	$\in$ {FLO, INT, CON}
hashref	Hash_entry	for *ctag* $\in$ {CON, STR}
flonum	double	for *ctag* = INT
intnum	flonum	for *ctag* = FLO

This translates to the following C data structure:

```
struct Indiv_entry
 {
 struct Indiv_entry *next;
 tag ctag;
 union
  {
  Hash_entry *hashref;
  long intnum;
  float flonum;
  } u;
 };
```

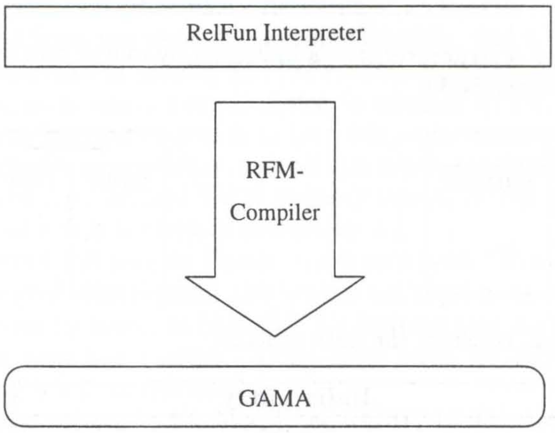
Further, we will give some pseudo code fragments of some pieces of the implementation. The syntax will be Algol-like and should be easy to understand. For pointer handling, the C '->' and '*' symbols will be used.

3 Overview

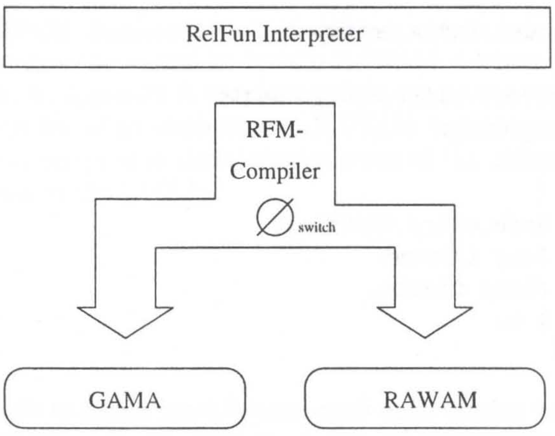
The RAWAM can be regarded as an alternative emulator for the Relational-Functional Machine (RFM, [BEH+96]). It is designed as an alternative for the RFM system's WAM emulator, GWAM. GWAM itself is embedded into an integrative platform, called GAMA, which, as a part of the RFM, supports GWAM, as well as another abstract machine, called LLAMA (GAMA and LLAMA are in detail described in [Sin95]).
Because the GAMA platform is not capable to maintain C-implemented programs, it cannot be used for the additional integration of the RAWAM. Therefore, the RAWAM is an execution environment of its own right; in fact, it must provide functionality not only of the GWAM, but also of the GAMA, and, by parts, of the LLAMA. The latter is because of the RFM's compilation logic which for efficiency reasons shifts some tasks from the GWAM to the LLAMA.
Without respect to the LLAMA part, RFM decomposes into three large parts:

- the RELFUN interpreter,
- the RFM compiler,
- the GAMA including the GWAM.

The data flow between these components can be seen top-down:



GAMA gets its data - WAM or LL code of declarative RelFun programs - via the compiler from the interpreter, where the RelFun interpreter also can be seen as completely independent from the rest of the RFM.
The RAWAM fits into this environment residing at the same level as the GAMA. This is realized by a switch that allows to use either of both components:



3.1 File Interface

Because of simpleness and portability, the communication between the RelFun compiler and the RAWAM is realized through a simple file interface. The communication consists of mutually creation of semaphore files associated to RelFun and the RAWAM, respectively, and a data-interchange file. The files are created as needed in the UNIX `/tmp/` directory.

3.2 What RAWAM gets from the RFM Compiler

The RAWAM communicates with the RFM via a file interface (see 3.1), where the contents are exchanged in a so-called interchange file, whose protocol distinguishes different types of data:

1. queries
2. `more` requests
3. program code
4. quit
5. result data

The first four are data created by the RFM part of RelFun, the fourth is the result data of the RAWAM. The RFM-generated data is distinguished by the tags `_Q`, `_M`, `_C`, and `_X`, respectively. The file is build up as follows:

1. A query, e.g., `X .= foo(A,5) & +(A,3)`, looks like this:

```
_Q
_QUERY/2
(
...  % some content omitted
)
```

The query consists of the tag `_Q` and WAM code for exactly one clause which is always called `QUERY/n`, where n is the number of distinct variables that occur in the query. In our example n is two, corresponding to the variables `X` and `A`.

2. This is a `more` request:

```
_M
```

3. Program code consists of RelFun clauses transformed to WAM code by the RFM compiler, and is submitted to the RAWAM in the form

1. the tag `_C`
2. sort data (cf. subsection 3.2.1)
3. an arbitrary number of WAM routines (cf. subsection 3.2.2)

4. This causes the RAWAM to quit:

```
_X
```

3.2.1 Sort Data

As mentioned above, sort data is contained in a LISP list and exactly of the format as described in [Hal95]. The structure of the sort data is given by the following grammar:

```
sortdata          ::= NIL | ( {sortspecification}+ )
sortspecification ::= ( sortname subsumeslist subsumes*list
                        individualslist individuals*list )
sortname          ::= identifier
subsumeslist      ::= ( SUBSUMES {identifier}* )
subsumes*list     ::= ( SUBSUMES* {identifier}* )
individualslist   ::= ( INDIVIDUALS {identifier}* )
individuals*list  ::= ( INDIVIDUALS* {identifier}* )
```

For more details refer to subsection 8.2.1.

3.2.2 WAM Routines

The input for RAWAM reflects the list oriented data representation of the LISP-based RFM system. Consider for example the following RelFun function:

```
nth([First|_],1) :& First.
nth([_|Rest],N) :& nth(Rest,1-(N)).
```

nth selects and returns the n-th element of a list. Compilation yields:

```
nth/2

((set_index_number 1)
 (switch_on_term nil nil "label126" nil "label126")
 "label126"
 (set_index_number 2)
 (switch_on_term "label132" 2 2 2 "label127")
 "label132"
 (switch_on_constant 1 ((1 "label127")) 2)
 "label127"
 (try 1 2)
 (trust 2 2)
 1
 (get_constant 1 2)
 (get_list 1)
 (unify_x_variable 3)
 (unify_x_variable 4)
 (put_x_value 3 1)
 (proceed)
 2
 (allocate 1)
 (get_x_variable 3 2)
 (get_list 1
 (unify_x_variable 4)
 (unify_y_variable 1)
 (put_x_value 3 1)
 (cl-func 1- 1)
 (get_x_variable 2 1)
 (put_y_value 1 1)
 (deallocate)
 (execute nth/2) )
```

The structure is much like in [GLLO85] and [AK91], with small differences and extensions, respectively. We can highlight here some basic features of RFM-generated WAM code:

- All clauses of same name and arity (here `nth` with arity 2) are put together into one WAM routine labeled by `name/arity`.
- The routine itself is a list containing WAM instructions and labels.
- The WAM instructions are represented as lists. The first element of a list is the instruction name and possible further elements denote the instruction's arguments such as, e.g., register numbers.
- If the indexing option of the RFM compiler is set, the first group of instructions forms the indexing part (see B.1.6), together with automatically generated indexing labels of the form `"label`*n*`"`.
- The actual WAM code for the clauses follows. Each WAM representation of a clause starts with a label which is just a number, starting from 0, ascending in the order in which the clauses occur in the RelFun source.
- An extension to the standard WAM instruction set is the `cl-func` instruction for calling a LISP function; here `1-` is called to decrement the contents of register X_1 by one.

The RFM system extends the basic WAM capabilities mostly not by adding additional instructions, but via WAM builtin functions. The only intrinsic extensions have been done in the indexing part and by calling LISP builtins via `cl-func`, `cl-relf`, `cl-extra`, and `cl-pred`. The RAWAM does currently not support this calling scheme. The most important LISP builtins, as arithmetics, are directly implemented as WAM commands without any calling indirection. For arithmetic commands see B.2.1; more special builtin reimplementations are denoted in the **Special Commands** paragraph of subsection B.2.2. These are substitutes for RFM builtin calls, e.g. `(call type/2)` is the same as the RAWAM instruction `(type)`.

3.3 Supported RelFun Language and Compiler Features

We list now the capabilities of the RAWAM.

3.3.1 Basic Instruction Set and Data Types

The RAWAM supports all instructions given in Aït-Kaci, except the `set...` instructions, because the RFM compilation knows and uses only their `unify...` counterparts.
Also, the `..._constant`-type instructions have a slightly different semantics. This is because the RAWAM distinguishes three types of constants: alphanumeric, integer, and floating point (we exclude here the empty list `[]`). So, a `..._constant`-type instruction can have only alphanumeric arguments, and for integer, respectively floating point arguments there are corresponding `..._int` and `..._float` instructions. For example, there exists not only a `put_constant` instruction, but there are also a `put_int` and a `put_float` instruction.
To execute RFM compiled programs, which know only `..._constant`-type instructions, correct, the RAWAM assembler analyses the arguments of a `..._constant` instruction and replaces it by the appropriate one.

3.3.2 Generalized Indexing

For the generalized indexing, as described in [Sin93], the RAWAM has, as the GWAM, an `IX` indexing register and provides the instructions `set_index_number` and `switch_on_term_n`, where the last one again is special, see appendix B.2.3.

3.3.3 Cut

RAWAM is capable of handling two different types of `cut` operators, one being the Aït-Kaci Y-Register `cut`, the other the GWAM `cut` which reserves a fixed slot on every stack frame. These two are distinguished by the occurence of either `get_level` or `save_cut_pointer`; if the former occurs in a clause, the `cut`s in the clause are assumed to be of the Aït-Kaci type, and the RAWAM `cut` instruction will be used, and if the latter occurs, `gama_cut` will be used.

Notice, that there will be no memory gain by using only the Aït-Kaci type `cut`, the stack slot will always be used.

3.3.4 Higher Order

A restricted higher order can be done by the `apply` instruction. It expects in the X_1register the name of a WAM routine to be called and in the X_2register a list containing the arguments which have to be given to the function. Currently `apply` does `not` bind a free variable in X_1 to a function's name. The X_1 argument always hat to be bounded to some constant.

3.3.5 Arithmetics

The RAWAM supports various arithmetic and comparison commands. All of them are listed in appendix B.2.1.

3.3.6 Types

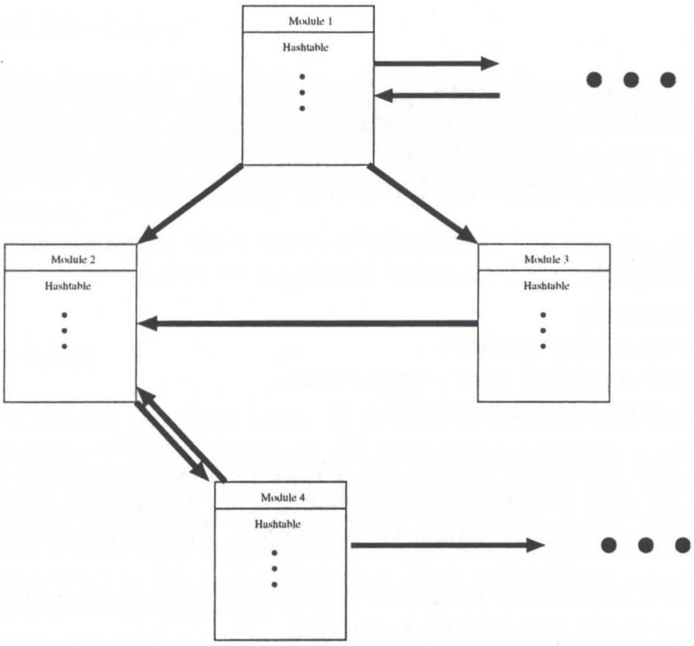
Types are implemented along the lines of the static sort model in [Hal95]. Builtin sorts are `$numberp`, `$atomp`, `$symbolp`, `$stringp`, `$floatp`, `$integerp`, `$evenp`, and `$oddp`.

3.3.7 Extra Instructions

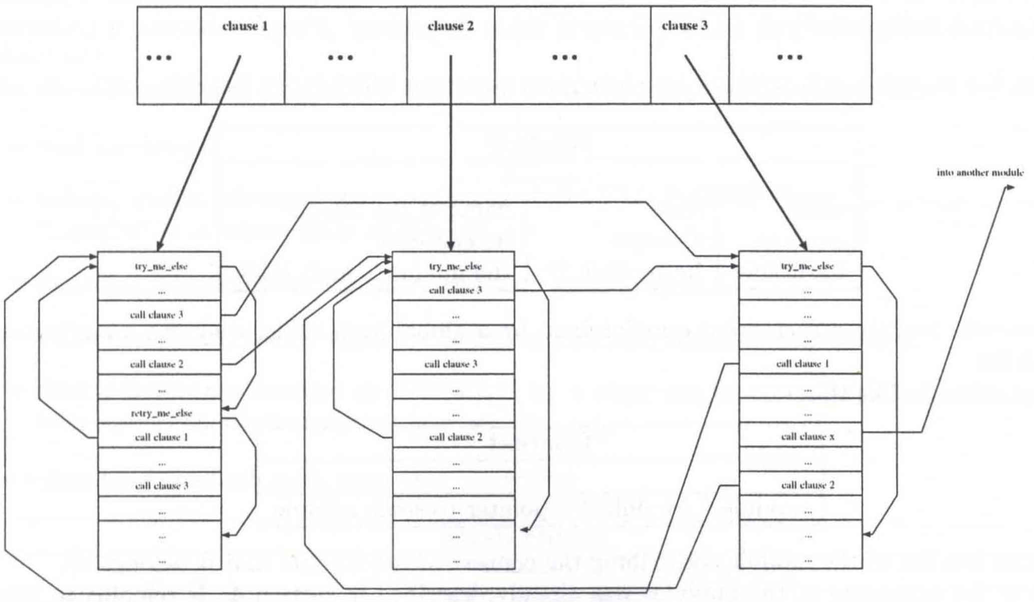
The extra instructions are the ones listed in the appendices B.2.2 (paragraph **Flow Control Instructions**) and B.2.3. The first ones are used for simplified flow control and usually not generated by any compiler but inserted by the assembler at appropriate places. The latter ones are compounds of combinations of standard instructions which often occur and are executed faster than the single instructions would be; see subsection 8.3.

4 Global Organisation

The RAWAM's main loop branches in two different states of operation: first, assembling WAM code coming from RFM and second, executing an assembled program, i.e. working as an abstract machine. The first part is a simple syntax-cruncher, transforming each RelFun clause into a byte code array; this is described in detail in section 8. The latter part is a bit more complicated. The topmost organisation structure consists of modules, described in section 5. Everything which is contained in one module, is organized in the module's hash table; this concerns literal constants as well as names of structures and clauses. In particular, each clause's byte code is attached to its entry in the hash table. Schematically, we have some general module organisation:



Here the boxes denote a module and the arrows denote contextual dependences between them, see section 5.
Each module has a hash table containing information for each literal found during the assembling process. In the simplest case, an entry in a hash table contains a symbol's name and arity. For sort specifiers, global variables, and clause names, additional data structures are created and linked by pointers. The execution scheme for clauses is as follows: as shown in the picture below, all clauses stored in the hash table have their own byte code array; in each `call/execute` or indexing instruction, a pointer is encoded. This pointer points to the very beginning of the byte code of the clause to be called.



If a query is given, the execution starts in the query's special byte code array, which is not contained in any hash table. The WAM `P` register, which is a pointer to byte code cells (of type `Codecell *`, see section 8.2), is set to the very first entry in this array. During execution, `P` is increased within the array, or set to a new array by `call/execute` commands.

5 Modules

The RAWAM module system is intended to provide static scoping domains for RelFun programs. Names are once resolved during compilation of a new database and remain statically until a new database is compiled. For details see [Her95]. Currently, it is not possible to do backtracking across contexts; this is also a feature which the RFM compiler lacks.

5.1 Using Modules

```
void init_modules(void);
Module *create_module(char *name);
void set_module_context(char *mod, char *ctxt);
static Module *find_module(char *name);
```

There are currently four functions for using the RAWAM's module system as follows:
`init_modules` must be used to initialize the module system. It creates a global hashing table in which references to the system's modules are maintained. The global variable `Modules` refers to this hash table.
`create_module` enters a module into the global module-hash table and names it after the given argument. A reference to the new module is returned.
`set_module_context` sets a module as context to another module. Both are refered by their names.
`find_module` searches a module whose name is the given one in the hash table and returns it.

5.2 Semantics of Modules

A RAWAM module consists of the following data:

1. the module name string,
2. a linked list specifying the module's contexts,
3. a hash table (see 6).

Hence, the module's informations are stored in a structure which looks like this:

Module		
$\longrightarrow$		
name	char *	name of the module
context	Context	its contexts
hashtable	hash_entry **	the module's hash table

Because the `Module` structures are maintained by a global hash table, they are organized as linked list.
A context looks like this:

Context		
$\longrightarrow$		
module	Module *	pointer to some module

`Context` is a list of the modules describing the context of the module that it belongs to.
Most of the semantics of the modules was already described in section 4. It remains to note that the structure of the context interdependencies of the modules are important in two places, which both use hashing: the assembling process and higher order calls via `apply`.

5.3 Standard Modules

At startup the following modules are created: `rawam-global`, `prelude`, and `workspace`.
`workspace` is the module where programs coming from the RFM compiler by default are compiled into. Its context is set to both, `rawam-global` and `prelude`.
`prelude` contains some predefined WAM routines, such as `member` or `tupof`. Its context is at startup `rawam-global`.
`rawam-global` should be set to the context of all modules. It contains names of important constants, such as `true` and `false`, which should be retrievable from each module.

6 Hash Tables

Associated to each module is a hash table. It is used to store all non-numeric symbols that occur during assembling of WAM programs. All data relevant for one symbol is put in a data structure of type `Hash_entry` which is built to form a linked list (see below). A hash table is an array of pointers to linked lists of `Hash_entry` structures. If the tokenizer (cf sect. 8.1) recognizes a symbol, a hash value is generated from its ASCII representation. This hash value determines a certain entry in the hash table array. Hash collision are avoided by appending a new symbol at the end of the linked list.
If one symbol denotes several distinguished type of data, these are not necessarily stored separately. This is only the case if it denotes function/structure names of different arity. E.g. if we consider the program

```
a(X).
a() :& a.
```

then the symbol a occurs three times: as name of functions of arity 1 and 0, respectively, and as a constant name. All three of them are by the hashing function mapped onto the same array index in the hash table. But, there are only two distinct entries in the linked list for the symbol a. One is the entry for the function name `a/1`, the other for the function name `a/0`. Because a constant is fully represented by its name, it is possible to represent it for the later steps of the assembling process also by `a/0`, ignoring all other properties which may belong to the *function* symbol `a/0`.
Now, the different types of symbols, which the RAWAM assembler recognizes, are:

- Sort identifiers.
- Labels, global labels of type `pred/n` as well as local labels which are strings of type `"label`*n*`"` or numbers (also as strings).
- Functors, which are constants associated to some arity.
- Non-numeric constants, which are stored as structure functors with arity 0.
- Global labels are handled as if it would be a structure functor; they carry additional information about the function (see below).

The contents of one hash table entry are:

Hash_entry		
	$\longrightarrow$	
name	char *	name of identifier
arity	short	arity of identifier (if needed)
stringconstant	bool	true, if name is string constant ("...")
sort	Sortbase_entry *	pointer to sort data
coderef	Codecell *	pointer to WAM instructions
domexc	DomExcTag	tag if name denotes exc/dom structure
globalvar_data	Heapcell *	data for global variable's contents
globalvar_size	long	size of global variable's data in terms of heap cells

7 The Byte Code

The byte code instructions are represented as elements whose type is a C-union `Codecell` which is capable to contain all possible data needed for byte code instructions:

Codecell		
tags	BYTE[4]	standard byte code format for short instructions
ctag	tag	for `switch_on_constant`
hashref	Hash_entry *	reference to alphanumeric constant
intnum	long	integer constant
flonum	float	floating point constant
coderef	Codecell *	jump destination

Each RAWAM instruction consists of a sequence of these `Codecells`. The first `Codecell` of an instruction always uses the `Codecell.tags` entries. `Codecell.tags[0]` contains the byte code (in the very sense of the word: a byte-sized number which encodes the instruction); `Codecell.tags[1]` and `Codecell.tags[2]` contain possible X- and Y-register denoting numbers, and `Codecell.tags[3]` nearly always contains the number of `Codecells` which follow the first one.
The only exceptions on the contents of `Codecell.tags[3]` are the `switch_on_constant` and `switch_on_structure` instructions, whose lengths may exceed the range of a byte-sized integer. Most instruction consist of only one `Codecell`. The exceptions are as follows, ordered by size:

Two-Byte Instructions:

- `put_structure`, `get_structure`, `put_constant`, `get_constant`, `unify_constant`: use the `Codecell.hashref` entry of the second `Codecell`.
- `put_int, get_int, unify_int`: use `Codecell.intnum`.
- `put_float, get_float, unify_float`: use `Codecell.flonum`
- `try_me_else, retry_me_else, try, trust, call, execute`: need precisely one jump destination as their second `Codecell`

Six-Byte Instructions: The `switch_on_term` is of the form

```
switch_on_term label1,label2,label3,label4,label5
```

where `label1`, ..., `label5` are some local labels or `NIL`. Therefore, `switch_on_term` needs six `Codecells` filled at the places `Codecell.coderef`. If `NIL` is given, the value there is set to zero. The `label`*n* entries correspond in the given order for indexing of WAM data types *floating point number*, *integer number*, *alphanumeric constant*, *structure*, and *list*,

Variable-Byte Sized Instructions:

- `switch_on_structure`: this instruction has, syntactically, the form

  ```
  switch_on_structure n,str1,label1,...,strn,labeln,labeln+1
  ```

  For this, `switch_on_structure` needs 2n + 3 `Codecells`, the first as usual, the second carries the value `n` in `Codecell.intnum`, the next 2n contain alternately first, a value in `Codecell hashref`, which denotes a structure identifier in a hash table (see below), and, second,a jump destination in `Codecell.coderef`. The last `Codecell` contains also a jump destination for the indexing cases that are not explicitely covered. This jump destination may also be zero, causing a failure.

- `switch_on_constant`: this instruction has the form

  ```
  switch_on _constant n,c1,label1,...,cn,labeln,labeln+1
  ```

switch_on_constant differs from that in [AK91] in the way that it distinguishes between integer, floating point, and alphanumeric constants; therefore it needs 3n + 3 Codecells, which are organized analogously to those for switch_on_structure; but the constant-jump destination pairs are here represented by Codecell triples, whose first member uses Codecell.tag, which contains one of the values CON, FLO, or INT, denoting the type of the triple's middle element, being, consequently, in one of Codecell.hashref, Codecell.intnum, or Codecell.flonum. The third element of each triple then contains finally the jump destination.

8 Generation of the Internal Program Representation

The generation of the internal program representation proceeds in several - not necessarily disjoint - steps. The rough scheme is as follows:
First, the incoming ASCII data is tokenized. The tokenized data is stored in a flexible list representation which allows easy manipulation. Simultaneously, jump destinations and labels are collected. Second, a pattern search is performed on this list which replaces groups of adjacent instructions by equivalent, single instructions. Third, the resulting list is transformed into a more rigid array representation and jump destinations are resolved.

8.1 Tokenizer

```
Token nexttoken(FILE *file, bool sortbase, Module *module);
```

The tokenizer is accessed via the routine nexttoken, which is called with three arguments:

1. a FILE pointer pointing to the input file (see sects. 3.1, 3.2),
2. a boolean, whose value must be true, if the file pointer points into a sortbase declaration part,
3. a pointer to the current module.

The tokenizer is basically a very simple DFA in the sense of, e.g., [ASU86]. If invoked by the assembler (cf. subsection 8.2), it scans the incoming (ASCII-)symbol stream until a valid token is recognized, or an invalid symbol sequence occurs.
Valid tokens can be

- global labels of the form name/arity (as, e.g., nth/2 in the example in subsection 3.2.2)
- local labels inside of a WAM routine, which may be 1. integer numbers, or, 2. of the form "labelXXX" where XXX is some integer.
- WAM commands of the form (name arg1, arg2, ...), same as shown in appendix B.

As mentioned in subsection 3.2.2, the commands in the RFM-generated WAM source are not necessarily mapped one-to-one to appropriate RAWAM instructions. So, e.g. get_constant is translated into one of get_constant, get_int, and get_float, depending of the type of its arguments (note that get_nil is not incorporated into this transformation, so that it is not translated from get_constant [],X1). The instructions which are diversificated this way are: get_constant, put_constant, and unify_constant, just by replacing of the ..._constant suffix by ..._float or ..._int. Together with this transformations, the switch_on_constant instruction is 'stretched' in the way that it gets two more argument positions for integer and floating point constants, which actually are filled with the same value as the by the *constant* argument given by the compiler has.
Second, certain calls and executes, respectively, are replaced by RAWAM instructions, e.g. (call type 1) is replaced simply by the command (type). Analogously all commands in subsection B.2.2 are replacements of calls to former GWAM builtin routines with the same name.

Third, the GWAM commands `cl-func`, `cl-pred`, `cl-relf`, and `cl-extra` are replaced by RAWAM instructions named in spirit of their arguments. For instance, `(cl-func + 3)` is replaced by the instruction `(addn 3)`, and `(cl-func + 2)` is replaced by `(add2)`. The former adds the contents of the registers X_1, X_2, and X_3 and stores the result in X_1. The latter does the same for the registers X_1 and X_2. This concerns mainly the arithmetic commands, shown in the appendix B.2.2.
The tokenizer is context sensitive in the manner that it automatically determines which of the two supported `cuts` (cf. 3.3) to use by the occurence of either `get_level` or `save_cut_pointer`; if the former occurs in a clause, the `cuts` in the clause are assumed to be of the Aït-Kaci type, and the RAWAM `cut` instruction will be used, and if the latter occurs, `gama_cut` will be used.
The tokenizer makes use of two sorts of data. The first data is a hash table for quickly recognizing instructions from given strings and associating them with their byte code, which is used for both, assembling and program execution (this hash table is once and for all initialized during program startup by the contents of an array called `wammcommands` containing pairwise instruction names and corresponding byte code numbers).
The second data are tokens. If a valid token is recognized, `nexttoken` will return the instructions data in a `Token` structure:

Token		
command	id	token identifier
reg1	short int	first register argument
reg2	short int	second register argument
codelength	short *	number of needed codecells
intnum	long	integer value
flonum	double	float value
hashref	hashref *	reference to some name
string	bool	true if *name* denotes a string constant
name	char *	name of constant identifier
sublabel	char *	label identifier
L	char *[5]	names of jump destinations for `switch_on_term`
table	indextable *	table for indexing

In detail: `command` contains an integer denoting the instruction's byte code, `reg1` and `reg2` contain numbers for registers. If the instruction needs only one register, always `reg1` is used. `codelength` contains the number of needed `Codecells` for the instruction. The first four entries usually represent the content of an instruction's first `Codecell`.
`intnum`, `flonum`, and `hashref` contain a constant's value and the reference to its name, respectively. When assembling WAM routines, alphanumeric constants are *immediately* stored by the tokenizer in the current module's hash table and are further referenced only via the reference there. If sort data is assembled, the constants are stored in the hash table by the assembler, not the tokenizer, because of the slightly different use of constants (see below). In this case, a reference to the constant's name string will be stored in `name`. Further information needed for the builtin sort `stringp`, is the distinction whether an alphanumeric constant is a string constant (`''something like this''`) or not. Then, `string` will have the value `true`.
If a local label is found (e.g. as instruction's argument as for `retry`), its name is not put into a hash table, but given back as string in `sublabel`. Analogously, `switch_on_term` needs five labels as arguments.
Last but not least, the indexing table is needed for the commands `switch_on_constant` and `switch_on_structure`; it is an array of entries of the following type:

indextable		
ctag	tag	∈ {`FLO`, `INT`, `CON`} if Token.command = `switch_on_constant` ∈ { `STR` } if Token.command = `switch_on_structure`
he	Hash_entry	for *ctag* ∈ {`CON`, `STR`}
flonum	double	for *ctag* = `INT`
intnum	flonum	for *ctag* = `FLO`
label	char *	name of local label related to indexing entry

Here to each pair of constant, the constant, the type of the constant and the name of the label to jump to are put together.

8.2 Assembler

```
void getcode(FILE *file, Module *module);
void getquery(FILE *file, Module *module);
static void getsortbase(FILE *file, Module *module);
```

The assembler consists of three parts:

- the function `getcode` for assembling a whole WAM program from a file
- the function `getsortbase` for getting sortbase information from a WAM program
- the function `getquery` for getting a query

Their arguments are always a reference to the input file and to the module into which all data have to be stored.

8.2.1 `getsortbase`

`getsortbase` parses the sort data coming from the RFM compiler. The sort data follows the grammar given in subsection 3.2.1, where the `(SUBSUMES ...)` and the `(INDIVIDUALS ...)` parts are ignored (these parts are artefacts from the sort processing in [Hal95] which contain only redundant data for our purposes). All not-ignored identifiers are stored in the given module's hash table. If a `sortname` of a `sortspecification` is stored in a module's hash table, a `Sortbase_entry` structure is created and a pointer is set in the `sortnames` `Hash_entry.sort` entry:

Sortbase_entry		
hashref	Hash_entry *	back-pointer to corresponding hash table entry
ctag	ctag	type of constant
subsorts	Subsumes_entry *	subsumes sorts
sortindividuals	Indiv_entry *	individuals of sort

The first entry in the `Sortbase_entry` structure points back to its hash table reference; this is used for `glb` calculations. The second entry denotes the type of constant it was originally recognized, either as integer, floating point, or alphanumeric. Note that a sort's name is always converted into a string.
`Sortbase_entry.subsorts` and `Sortbase_entry.sortindividuals` both are pointer to linked lists containing all subsumed sorts and individuals contained in the sort, respectively (see below).
If a `(SUBSUMES ...)` part is parsed, all occuring constants are put into the hash table (if they were not yet there) and their references are stored into a linked list of type `subsume_entry`, a pointer of which is stored into `Sortbase_entry.subsorts`:

subsumes_entry		
$\longrightarrow$		
hashref	Hash_entry *	reference to subsumed sort

Analogously, a `(INDIVIDUALS ...)` part is parsed and the constants found there refered by a linked list of type `Indiv_entry`, being pointed to by `Sortbase_entry.sortindividuals`:

Indiv_entry		
$\longrightarrow$		
ctag	tag	$\in$ {`FLO`, `INT`, `CON`}
hashref	Hash_entry	for *ctag* $\in$ {`CON`, `STR`}
flonum	double	for *ctag* = `INT`
intnum	flonum	for *ctag* = `FLO`

It is easy to see, that these data structures are only C-data structure copies of the corresponding LISP lists.

8.2.2 getcode and getquery

```
Codecell *writecode(Codeconstruct *first, long *count, jumptable_entry **je);
subjumptable_entry *lookup_label(char *name);
subjumptable_entry *enter_label(char *name);
void cleansubjumptable(void);
```

Logically, `getquery` is a special case of `getcode`, because a query could be seen as a WAM program consisting of only one clause. So the two routines are splitted mainly for historical reasons. Further, `getsortbase` is used only by the function `getcode` at the very beginning for parsing sort data (see subsection 3.2.1).
The processing scheme is as follows: the stream of tokens coming from the tokenizer is analyzed and transformed into separate linked lists of `Codeconstruct` structures, one linked list for each WAM routine. After a WAM routine is completely read in and brought into this list representation, the next steps are a bit of optimization (cf. 8.3), resolving the jump addresses, and, at last, the list is transformed into a byte code array.
The intermediate linked lists representation consists of `Codeconstruct` structures:

Codeconstruct		
com	Codecell	represented Codecell
pass1	subjumptable_entry	preliminary jump destination
pass2	Codecell *	associated address in Codecell array

The first entry, `Codeconstruct.com`, is a prototype of the `Codecell` (i.e. bytecode instruction) which finally will be generated. In most cases, this prototype will not be altered. Execeptions are `Codecells` which will contain memory addresses which are not fixed in this state, as, e.g., jump destinations coming from `call` or indexing instructions. The other two entries are needed for the creation of the byte code and the resolving of jump destinations.
The data created during the assembling are the following:

- Hash table entries for each newly occuring symbol.

This is already done by the tokenizer and shall not further be considered.

- Linked `Codeconstruct` lists for each WAM routine which is assembled.
- Linked `subjumptable_entry` lists (see below), keeping track of local jump destinations needed by indexing and backtracking.
- Byte code arrays for each WAM routine.

These are created for each WAM routine which is assembled. The first two are just for intermediate use, where the byte code arrays are the very results of the assembling.

- A linked `jumptable_entry` list which keeps track of each `call` and `execute`; it is used for resolving the global jumps.

This data structure survives the assembling of single WAM routines and is used to insert the correct jump destinations into the byte code after all routines have been assembled.
The `pass1` and `pass2` entries of a `Codeconstruct` are used as follows: if a label of a subroutine of a WAM routine is read, either direct as label or as an argument of an, e.g., indexing, instruction, this is stored in a hash table of `subjumptable_entry` structures:

subjumptable_entry		
	$\longrightarrow$	
name	char *	name of jump label
pass1	Codeconstruct	first Codeconstruct cell of routine which belongs to the label
pass2	Codecell	first Codecell of routine which belongs to the label

These are created with the `enter_label` function and retrieved with `lookup_label`. The function `cleansubjumptable` erases the actual contents of the table. If a `Codeconstruct` refers to a certain label, being the first instruction after the label or belonging to an indexing or backtracking instruction, this hash table entry is stored in `Codecell.pass1`. If a sublabel is found as label and not as argument of an instruction, the `Codecell` structure of the first command following it is stored in `subjumptable_entry.pass1` to denote the entry point of this subroutine.
Constructing the `Codeconstructs` is the first assembler pass (relatively to each WAM routine, of course). The second is to copy them into an array of `Codecells` (and filter out some garbage `Codeconstructs` which may be created sometimes). The `Codeconstructs` are not yet thrown away, but the `Codeconstruct.pass2` entries are set to the corresponding `Codeconstructs` address. Also, the `subjumptable_entry.pass2` entries can now be set to the appropriate `Codecells`.
In a third pass, all the collected data are used to update all references to local jump destinations in the routine's `Codecell` array[1].
Simultaneously, in this last pass, the occuring `call` and `execute` instructions are checked, and for each a `jumptable_entry` structure is created, which associates the instruction's argument with an entry in the hash table, which should denote another WAM routine's name:

jumptable_entry		
$\longrightarrow$		
coderef	Codecell	refer to a `calls` or `executes` `Codecell`
hashref	Hash_entry	jump destination given by this label

The linked list of `jumptable_entry` structures will not be used until the whole WAM program is read in. After that, the assembler steps through this list and copies the `Hash_entrys` `coderef` (cf. 6) value into the `Codecell` denoted by `jumptable_entry.coderef`. In reminder of section 4, this denotes just the first instruction of a WAM routine stored in a module's hash table.

8.3 The Optimizer

```
Codeconstruct *optimize(Codeconstruct *code, long *count);
```

The optimizer runs through the `Codeconstruct` list of a WAM routine just before it is transformed to a `Codecell` array. It tries to find groups of instructions which can be comprehended to a single instruction. The optimizer consists of one function whose arguments should be a pointer to the first `Codeconstruct` structure of a WAM routine, as well as a pointer to an integer, in which the new number of the optimized list is returned. The function returns a pointer to the new list.
The routine steps through the list and searches for sublists, according to the second column in the table in appendix B.2.3. If one is found, the sublist is removed and replaced by the appropriate single instruction as in B.2.3. Note, that `optimize` reuses the `Codeconstructs` rather than allocating new ones. This is important because at this stage the jump destinations are not yet resolved, and the corresponding data structures, described in the previous subsection, have to stay valid.

9 WAM Registers

The RAWAM supports different kinds of registers, which we classify as follows:
There are registers for

- temporary use,
- permanent use,
- program execution,

[1]note that a `nil entry` in a indexing instruction's are automatically filled with the address of the `backtrack` command.

- indexing,
- time stamping,
- and memory management.

The first two correspond directly to the usual X- and Y-registers. The program execution registers are the known `P` and `CP` registers. The indexing register `IX` is completely the same as described in [Sin93] to support the generalized indexing scheme there. The time stamp register `Timestamp` holds the value needed for giving stack slots a unique mark to determine the order of creation, as described in 11.4.3.
"Memory management" comprises here the usual `H`, `HB`, `S`, `TR`, `E`, `B`, and `BO` registers, as well as - we must now anticipate section 11 - all global values which are needed to handle the corresponding `HST` structures. These are: `Heap`, `Stack`, and `Trail`, pointing to the very first `HST` structure of the identically named memory area ; `CurrHHeap`, `CurrBStack`, `CurrSHeap`, `CurrBOStack`, `CurrEStack`, and `CurrHBHeap`, to be read as, e.g., `HST` structure which manages the memory chunk where *Curr*ently the *H* register points into, belonging to the *Stack*; the analogous register for `TR` is, inconsequently, called `CurrTrail`. Then there are the registers `HeapBot` and `HeapTop`, which are set to `CurrHHeap->low.heap` and `CurrHHeap->high.heap`. Analogously `SHeapBot`, `SHeapTop`, `TrailTop`, `StackTop`, `StackBot` (topmost `HST` concerning E and B), `BOStackBot`, and `BOStackTop`.
For detailed information about the latter classes of registers, see section 11.

10 WAM Extensions

10.1 WAM Data Types

As in [AK91], heap cells are tagged structures. Besides the standard heap cells with tags `CON`, `STR`, `LIS`, `NIL`, `FUN`, `REF`, there are three additional tags `FLO`, `INT`, and `TYP`. This is because of the fact that the RAWAM distinguishes three types of constants, namely integers, floating point numbers, and alphanumeric constants, and because the RAWAM supports a sorted type system. As a consequence, the semantics of `CON` and `REF` is slightly different. A `CON`-tagged heap cell contains only alphanumeric constants, and no integers or floating points. On the other hand, a `REF` tagged cell never contains a self reference; a free variable is denoted by `TYP`, and it eventually contains a reference to its type. All the other tags are the same as described in [AK91].

10.2 Sorts

```
Heapcell *domexc(Heapcell *, Heapcell *);
Heapcell *exc_union(Heapcell *, Heapcell *);
Heapcell *intersect(Heapcell *, Heapcell *);
Heapcell *glb(Heapcell *, Heapcell *);
Heapcell *sort_to_dom(Sortbase_entry *);
Heapcell *sortglb(Sortbase_entry *, Sortbase_entry *);
void element_of(Heapcell *, Heapcell *);
Heapcell *intersectdombuiltin(Heapcell *, Hash_entry *);
Heapcell *intersectbuiltins(Hash_entry *, Hash_entry *);
```

The RAWAM supports, as the GWAM does, the static type model as described in [Hal95]. For this support the RAWAM design is changed as follows:

1. There exists a new type of heap cells identified by the tag TYP, as described in subsection 10.1.
2. There is an additional command `type`, which expects a sort in register X_2 and calculates its *glb* with the contents of register X_1.
3. The `bind` command has also been modified to calculate *glb*'s.

We give here patterns for bind and type; see for documentation the comments inside and for Heapcell refer to subsection 11.2.1:

```
// bind expects its left argument to be dereferenced, i.e <> REF and its right
// argument to be TYP

procedure bind(left : Heapcell *, right : Heapcell *)

// arguments:
// left  : reference to some heap cell
// right : reference to some heap cell which must contain a TYP

// non-nil heaptype means, this variable is typed:

 if right->u.heaptype <> 0 then
  begin
   if left->tags[0] = TYP then          // if left argument is a variable,
    begin                               // we may have to do sort calculations
     Trail_Var(left);

     if left->u.heaptype <> 0 then      // calculate glb
      left->u.heaptype = glb(right->u.heaptype, left->u.heaptype);

     else
      left->u.heaptype = right->u.heaptype;

     Trail_Var(right);
     right = <REF, left>;
     return;
    end;

// if left argument is some constant, it may be contained in
// the domain of the right:
   else if left->tags[0] = CON or left->tags[0] == INT or
           left->tags[0] = FLO then
    begin
     element_of(left, right->u.heaptype);

     if fail == true
      return;
    end;

   else
    begin
     fail = true;
     return;
    end;
  end;

// in this case, left may have some type, copy it:

 else if left->tags[0] = TYP then
  begin
   Trail_Var(right);
   right->u.heaptype = left->u.heaptype;
  end;
 Trail_Var(right);

 if left->tags[0] = TYP then
  left = <REF, right>;

 else
  *right = *left;
end bind;
```

The semantics of the type instruction is to expect a variable or constant in the X_1 register and a sort in the X_2 register, to compute the glb and to store the result in X_1, or to fail, if X_1contains a constant and it is not in the domain of the sort. The instruction looks as follows:

```
procedure type
 Deref(XReg(1), addr); // dereference contents of X1

// Check, if X2 contains type or dom/exc:

 if X2->tags[0] = CON or (X2->tags[0] = STR and
    X2->u.heapref->u.hashref->domexc <> 0) then
  begin

// do type calculations:

   if addr->tags[0] = TYP then    // if yes, then
    begin
     Trail_Var(addr)              // variable must be saved

     if addr->u.heaptype = 0 then  // if addr contains no type, it gets
      begin                        // a copy, which must be on the
       *H = *X2;                   // heap
       addr->u.heaptype = H;
       Inc_H(1);
      end;

     else                         // else compute glb
      addr->u.heaptype = glb(addr->u.heaptype, X2);
     end;

// if addr contains constant, glb is the question:
// contained or not contained?

    else if addr->tags[0] = CON or addr->tags[0] = INT or
         addr->tags[0] = FLO then
     element_of(addr, X2);
  end;

// couldn't perform type calculation:

 else
  fail = true;

// proceed:

 if fail = true then
  begin
   P = B[B[1]->intnum + 7]->coderef;
```

```
    fail = false;
   end;

  else
   P = P + 1;
 end type;
```

There are several functions to implement the *glb* calculations. Here is a list (see also the beginning of this subsection):

- glb is the general *glb* function which branches into the others.
- domexc calculates the intersection of a dom and a exc sort.
- exc_union forms the union of two exts.
- intersect intersects two doms.
- sort_to_dom transforms a user-defined finite sort into a dom.
- sortglb calculates the *mgu* of two user-defined sorts.
- element_of checks if a constant is contained in a given sort.
- intersectdombuiltin intersects a dom with a builtin sort.
- intersectbuiltins intersects builtin sorts.

Note that all functions except element_of and intersectbuiltins store their results on the heap.

10.3 Higher Order

The current higher order facilities are covered by the apply instruction. This instruction expects a constant in the X_1 register and a list of arguments to be given to the constant in X_2 . It does *not* work in the case that the X_1 register contains a free variable. Here is the pattern:

```
procedure apply
 Deref(X1, addr);         // dereference contents of X1

 he = addr->u.hashref;    // get its reference in the hash table

 if he = 0 then
  fail = true;

 else
  begin
   name = he->name;       // get name of function to be called
   Deref(XReg(2), list); // get list of arguments

   for(i = 1; list && Tag(list) == LIS; i++) // fill registers with arguments
    {
    list = list->u.heapref;
    *XReg(i) = *(list);
    list++;
    }

   he = lookup_proc(Current, name, i - 1);  // find routine with given arity

   if he = 0 then
    fail = true;
   else if he->coderef = 0 then
    fail = true;

   else
    begin
     if Argument1(P) = 0 then           // *** see in text below
      CP = P + 1;

     B0 = B;                            // call routine
     P = he->coderef;
     return;
    end
  end;

// proceed:

 if fail = true then
  begin
   P = B[B[1]->intnum + 7]->coderef;
   fail = false;
  end;

 else
  P = P + 1;
end apply;
```

`***`: Note that the RFM compilers implementation of higher order calls works via a call `(call apply/3 0)` or `(execute apply/3 0)`, so that the `apply` command has an additional parameter to update the `CP` register if it was called by `call`. If `(apply 0)` is used, this comes from `(call...)`, else it comes from `(execute...)`.

11 Heap, Stack, Trail, PDL

Memory areas needed for program execution are, as in [AK91], the so-called Heap, Stack, Trail, and PDL. The first three are manipulated directly by the WAM instructions; the PDL is used indirectly as the recursion stack of the **unify** function and is therefore implemented implicitely by the C stack.
In [AK91], heap, stack, and trail are organized in the way so that they are located in one large, connected memory area, where heap, stack, and trail are exactly in this order, counted from the lower to the higher addresses.
Unlike this, RAWAM organizes its memory in several smaller, position-independent memory units, which allows dynamic allocation of new memory, if necessary. Each data area is handled separately as a linked lists of memory blocks whose first element is accessed through the global pointer variables **Heap**, **Stack**, and **Trail**. These variables are pointers to the **HST** structure (**H**eap **S**tack **T**rail):

HST		
⟵ ⟶		
size	size_t	size of memory block
heap	Heapcell	pointer to low memory address
stack	Stackcell	pointer to low memory address
trail	Trailcell	pointer to low memory address
heap	Heapcell	pointer to high memory address
stack	Stackcell	pointer to high memory address
trail	Trailcell	pointer to high memory address

The **low** entry points to the first element of an array of heap-/stack-/trail- cells, and **high** points one above the highest element of this array. It always holds that *HST.low.stack + size == HST.high.stack*, and for any address *A*, *A* points into the heap-/stack-/trail-memory block, iff *HST.low.stack <= A < HST.high.stack* (respectively, *HST.low.heap, HST.low.trail* etc.).

11.1 General Handling of HST

We give a quick reference of which general functions are implemented to deal with HST structures. The data type memtype here can take one of the values HEAP, STACK, or TRAIL; memtype is used for clean type distinction, so that the functions below know if to use the heap, stack, or trail entries within the HST.

```
HST *init_hst(size_t size, memtype memtype)
```

Initializes an HST structure consisting of one block and of size size.

```
HST *new_hstchunk(size_t size, memtype memtype);
```

Allocates and initializes one HST structure together with a memory block of size size.

```
void remove_hstchunk(HST *h, memtype memtype);
```

Given an HST structure within a linked list of HST structures, remove_hstchunk removes this element out of the linked list.

```
void cleanup_hst(HST *h, memtype memtype);
```

Assumes that a linked HST lists starts with the HST given by the first argument and deallocates the whole linked list.

```
HST *append_hstchunk(HST *h, size_t size, memtype memtype);
```

If h represents the beginning of a linked HST list, a new element for the given size size is appended to it.

```
HST *insert_hstchunk(HST *hl, HST *ch);
```

Assumes that hl is contained in a linked list and inserts ch immediately before hl.

```
HST *insert_new_hstchunk(HST *h, size_t size, memtype memtype);
```

Same as insert_hstchunk, but allocates a new HST structure for the given size.

```
HST *hfind_hstchunk(Heapcell *h), *sfind_hstchunk(Stackcell *s),
    *tfind_hstchunk(Trailcell *t);
```

These functions return a pointer to the HST structure into whose memory h, s, or t respectively, points. These functions are specialized to search only in the HST lists belonging to heap, stack, or trail.

```
HST *hrelfind_hstchunk(Heapcell *h, HST *hl);
```

Searches and returns the HST structure in the linked list starting with hl into whose memory block h might point.

11.2 Heap

11.2.1 Data

One heap HST memory unit is merely an array of Heapcell structures. The size of one HST memory unit is given by the global constant HEAP_SIZE, and the maximal number of heap cells is given by the constant HEAP_MAX_SIZE (hence the maximum number of HST blocks for the heap can be HEAP_MAX_SIZE / HEAP_SIZE). The current number of allocated heap cells is stored in the variable Heapsize, which is used to compare the current size of the heap with the maximum allowed. Heapcells can be of different types; each type differs by the kind of data it contains. A Heapcell has two entries: its tag and its value. There exist the following tags and data associated to them:

- NONE if the Heapcell is unspecified
- FLO Heapcell contains floating point number
- INT Heapcell contains integer
- CON Heapcell contains literal constant
- STR Heapcell contains pointer to a WAM structure
- LIS Heapcell contains pointer to list element
- NIL Heapcell contains end of list symbol
- TYP Heapcell contains free variable, respectively points to type data
- FUN Heapcell contains functor symbol
- REF Heapcell contains reference to another Heapcell

The Heapcell is therefore defined by a C-structure having two entries, where the first entry contains the tag and the second the data. The first entry in principle would need only 4 bits, but because no efford is done for considering special data encoding, we have to stick to standard C data types; this means we need at least 1 byte, and if taken alignment within the structure implicitely into account, we easily arrive at 4 bytes. The data entry of a Heapcell now is realized by a appropriate union:

Heapcell		
tags	BYTE[4]	tags[0] holds type of heapcell, tags[1]..tags[3] are not used yet
flonum	double	floating point number
intnum	long int	long integer number
hashref	Hash_entry *	name of a constant (for FUN)
stackref	Stackcell *	pointer to stack
heapref	Heapcell *	pointer to heap (for REF, STR, LIS)
heaptype	Heapcell *	pointer to type information (for TYP)

11.2.2 Access

The data used for manipulating and using the heap is encoded in several ways. First, the implementation specific memory restrictions are handled, as described before, by the global values/variable HEAP_SIZE, HEAP_MAX_SIZE, Heapsize, and Heap. Second, the register set is extended from the usual H, HB, and S registers by the registers CurrHHeap, CurrSHeap, and CurrHBHeap. These are set to the HST memory blocks into which H, S, and HB, respectively, point. Further, the registers HeapBot and HeapTop are set to CurrHHeap->low.heap and CurrHHeap->high.heap. Analogously, SHeapBot contains CurrSHeap->low.heap, and SHeapTop contains CurrSHeap->high.heap.
One should notice, that all this registers formally are treated on the same level, but, in fact, the additional HST-related registers are only auxiliary registers for the standard registers H, S, and

HB. So it is convenient to provide access operations for the standard registers and encapsulate the use of the additional registers within these.
For the heap, there are four such operations, `Inc_H`, `Inc_S`, `Set_S`, and `Next_HCell`.
`Inc_H` and `Inc_S` are mostly identical, they are used for increasing the contents H and S registers by a given number:

```
procedure Inc_H(N : integer)

// arguments:
// N : increment H by N heap cells

 if H + N < HeapTop then                      // this is the uncritical case:
  H = H + N;                                  // simply increase H

 else                                         // in this case the boundary of
  begin                                       // the current HST block is reached
   NN = N;

// now it is to check, if we have to allocate a new entry in the
// heaps HST list:

   if CurrHeap->next = 0 then
    if Heapsize + HEAP_SIZE <= HEAP_MAX_SIZE then  // check for memory limit
     begin
      append_hstchunk(CurrHeap, HEAP_SIZE, HEAP);  // allocate new HST
      Heapsize = Heapsize + HEAP_SIZE;     // update allocated-memory index
     end;

    else                                      // no more memory -> fail
     begin
      fail = true;
      return;
     end;

// Now CurrHeap->next holds a valid HST structure

   CurrHeap = CurrHeap->next;

// Now we handle the extreme case: if HEAP_SIZE is set to some low
// value, the increase N may need to extend the heap by some more HSTs

   while NN > HEAP_SIZE do
    begin
     if CurrHeap->next = 0 then
      if Heapsize + HEAP_SIZE <= HEAP_MAX_SIZE then
       begin
        append_hstchunk(CurrHeap, HEAP_SIZE, HEAP);
        Heapsize = Heapsize + HEAP_SIZE;
       end;

      else
       begin
        fail = true;
        return;
       end;

     CurrHeap = CurrHeap->next;
     NN = NN - HEAP_SIZE;
    end;

// now update all related registers:

   HeapBot = CurrHeap->low.heap;
   H = HeapBot + (N - (HeapTop - H))
   HeapTop = CurrHeap->high.heap;
  end;
end Inc_H;
```

Inc_S does the same for the S register as Inc_H for the H register does, except that the check at the very beginning has to take respect to the fact that S in general does not point to the topmost heap element. So we give here the pattern without further comments:

```
procedure Inc_S(N : integer)

// arguments:
// N : increment S by N heap cells

 if (SHeapBot <= S + N) and (S + N < SHeapTop) then
  S = S + N;

 else
  begin
   NN = N;

   if CurrSHeap->next = 0 then
    if Heapsize + HEAP_SIZE <= HEAP_MAX_SIZE then
     begin
      append_hstchunk(CurrSHeap, HEAP_SIZE, HEAP);
      Heapsize = Heapsize + HEAP_SIZE;
     end;

    else
     begin
      fail = true;
      return;

   CurrSHeap = CurrSHeap->next;

   while NN > HEAP_SIZE do
    begin
     if CurrSHeap->next = 0 then
      if Heapsize + HEAP_SIZE <= HEAP_MAX_SIZE then
       begin
        append_hstchunk(CurrSHeap, HEAP_SIZE, HEAP);
        Heapsize = Heapsize + HEAP_SIZE;
       end;

      else
       begin
        fail = true;
        return;
       end;

     CurrSHeap = CurrSHeap->next;
     NN = NN - HEAP_SIZE;
    end;

   SHeapBot = CurrSHeap->low.heap;
   S = SHeapBot + (N - (SHeapTop - S)) % HEAP_SIZE;
   SHeapTop = CurrSHeap->high.heap;
  end;
end Inc_S;
```

Set_S sets the S-register to a given register and updates the auxiliary registers:

```
procedure Set_S(A : address)

// arguments:
// A : set S to this address

 hst = hfind_hstchunk(A);          // find the HST structure into whose memory
                                   // A points.
 if hst <> CurrSHeap then          // if this differs, update the auxiliary
  begin                            // registers
   CurrSHeap = hst;
   SHeapBot = hst->low.heap;
   SHeapTop = hst->high.heap;
  end;

 S = A;
end Set_S;
```

Next_HCell sets the argument HH to point to the Heapcell consecutive to the Heapcell H. H points into the memory block belonging to the HST CH.

```
procedure Next_HCell(HC : address, CH : address, HH : address)

// arguments:
// HC : address to heap cell whose successor must be found
// CH : HST which contains HC
// HH : the result will be copied here

 if HC + 1 < CH->high.heap then              // the easy case
  HH = HC + 1;

 else                                // boundary of HST memory reached
  begin

// now it is to check, if we have to allocate a new entry in the
// heaps HST list:

    if CH->next = 0 then              // allocate new HST
     if Heapsize + HEAP_SIZE <= HEAP_MAX_SIZE then
      begin
       append_hstchunk(CH, HEAP_SIZE, HEAP);
       Heapsize = Heapsize + HEAP_SIZE;
      end;

     else                                    // no more memory -> fail
      begin
       fail = true;
       return;
      end;

   HH = CH->next->low.heap;         // Set HH to the first new Heapcell
  end;
end Next_HCell;
```

11.3 Trail

The trail's data is, as for the heap, represented by several types of data. There are the global values `STACK_SIZE` and `STACK_MAX_SIZE`, determining the number of stack cells managed by the trail's `HST`s and their maximal number; the trail's current size is determined by the global variable `Trailsize`. The `HST`s are a linked list pointed to by the global variable `Trail`. The trailing register `TR` is as usual, and the trail's auxiliary registers are `CurrTrail`, holding the address of the `HST` into whose memory `TR` points, and `TrailTop`, pointing to the top address of this `HST`'s memory.

11.3.1 Trailcells

To support RelFun's typed variables (see [Hal95]), RAWAM's trail is modified in the way that it is not an array of pointers Heapcells (or, more generically, as addresses of heap cells), but as array of pairs of pointers to Heapcells. One memory segment of a trail-**HST** structure looks like:

Trail	
(Heapcell *)[2]	couple of pointers to heap cells
(Heapcell *)[2]	...
...	
...	

11.3.2 Access

In [AK91], the trail is used by the WAM instructions via three routines: *trail*, *unwind_trail*, and *tidy_trail*. RAWAM supports only the first two, here called `Trail_Var` and `Unwind_Trail`, implemented as macros (as C substitute for inline functions). `Trail_Var` works analogous to the corresponding pattern in [AK91], but `Unwind_Trail` has to take care of the order in which it restores the free variables on the heap. This is important because of the sort unification, whose results are also stored in the trail, and which have to be recovered in reverse time order. However, `Unwind_Trail` solves this in a rather trivial way: restore the variables downwarts from the top of trail, instead of upwards. For the same reason, *tidy_trail* was modified to equate its efficiency benefits under this circumstances: nil.
Here is the RAWAM-modified pattern for `Trail_Var`:

```
procedure Trail_Var(T : address)

// arguments:
// T : variable which to trail

// first, search the heap-HST T belongs to, it should be after CurrHBHeap:

 hst = CurrHBHeap;

 while hst <> 0 and not(T >= hst->low.heap and t < hst->high.heap) do
  hst = hst->next;

// now check, if the variable has to trailed:
 if ((hst = 0) and not(in(T,Current Stackframe) and T >= B))
    or ((hst <> 0) and (T < HB)) then
  begin
// fill the topmost entry in the trail with the variable's data:
   (*TR)[0] = T;
   (*TR)[1] = T->u.heaptype;

// Try to increase TR or check if top of trail-HST is reached and allocate
// new one if necessary:

   if TR < TrailTop - 1 then
    TR++;

   else
    begin
     if CurrTrail->next then
      if Trailsize + TRAIL_SIZE <= TRAIL_MAX_SIZE then
       begin
        append_hstchunk(CurrTrail, TRAIL_SIZE, TRAIL);
        Trailsize = Trailsize + TRAIL_SIZE;
       end;

      else
       begin
        fail = true;
        return;
       end;

     // update registers:

     CurrTrail = CurrTrail->next;
     TrailTop = CurrTrail->high.trail;
     TR = CurrTrail->low.trail
    end;
end Trail_Var;
```

Unwind_Trail restores variables in the reverse order they were put onto the trail. In the simple case that all variables are contained in one HST memory block, this is straightforward. In the other case, several HST memory blocks may to have skipped.

```
procedure Unwind_Trail(A)

// arguments:
// A : trail address up to which to unwind

// easy case: simply loop through

 if (A < CurrTrail->high.trail) and
    (A >= CurrTrail->low.trail) then
  begin
   h = TR - 1;

   repeat
    *(*h)[0] = <TYP,(*h)[1]>;
    h = h - 1
   until h = A;
  end

// difficult case: restore all variables from the topmost trail HST. If
// the number of variables to restore is significantly larger than
// the size of the trail HST memory blocks, several of them may have to
// be fully released. Finally, everything is reduced to the easy case

 else
  begin
   hh = A;
   trl = tfind_hstchunk(A);          // find chunk to reach
   h = TR - 1;

   repeat                            // first, do entries in top HST block
    *(*h)[0] = <TYP,(*h)[1]>;
    h = h - 1
   until h = CurrTrail->low.trail;

   CurrTrail = CurrTrail->prev;      // go to previous HST block

   while CurrTrail <> trl do         // munge all intermediate HST blocks
    begin
     h = CurrTrail->high.trail - 1;

     while h >= Currtrail->low.trail do
      begin
      *(*h)[0] = <TYP,(*h)[1]>;
      end;

     CurrTrail = CurrTrail->prev
    end

   h = CurrTrail->high.trail - 1;

   repeat                           // do last HST block
    *(*h)[0] = <TYP,(*h)[1]>;
    h = h - 1
   until h = A;

// update remaining auxiliary registers:

   TrailTop = CurrTrail->high.trail
  end
end Unwind_Trail;
```

11.4 Stack

The stack contains choice point and environment frames, which are a bit enlarged, compared with [AK91], for the need of our enhanced memory management. As for the Heap, for the full representation and use of the stack, we need several kinds of data. There are the fixed constants STACK_SIZE and STACK_MAX_SIZE determining the size of the memory belonging to one HST, and the maximum amount of memory which may be occupied by the stack; the global variables Stack and Stacksize denote the linked HST list and the number of currently allocated Stackcells. Further, besides the known E, B, and B0 registers, there are some HST-valued registers associated to each of them:

- `CurrStack` contains a pointer to the `HST` into whose memory `B` currently points. The same do `CurrB0Stack` and `CurrEStack` for `B0` and `E`.
- `StackBot` and `StackTop` contain the lowest and highest memory addresses for the memory block of `CurrStack`. Same for `B0StackBot`, `B0StackTop`, and `CurrB0Stack`.

11.4.1 Layout

The layout of a choice point is as follows:

Offset	Content
0	Timestamp register
1	number n of saved Y-registers
2	Y-register 1
...	
n + 1	Y-register n
n + 2	E register
n + 3	CurrEStack register
n + 4	CP register
n + 5	B register
n + 6	CurrStack register
n + 7	next clause in case of failure
n + 8	TR register
n + 9	H register
n + 10	CurrHeap register
n + 11	B0 register
n + 12	CurrTrail register

Environment frame layout:

Offset	Content
0	Timestamp
1	E register
2	CP register
3	B0 register for cut
4	CurrEStack register
5	first Y-register
...	
4 + m	mth Y-register

11.4.2 Stackcells

A `Stackcell` can contain several kinds of data; there is no tagging needed, because there are fixed rules how stack slots are filled. Hence, a `Stackcell` is a union as follows:

Stackcell		
reg	Heapcell	for X-registers
stackref	Stackcell *	pointer to stack
intnum	unsigned long int	long integer number
coderef	Codecell *	pointer to program address
hstref	HST *	pointer to memory segment
trailref	Trailcell *	pointer to trail
heapref	Heapcell *	pointer to heap

11.4.3 Timestamps

To compare the creation time of environment frames and choice points, each of them has an additional slot for a time stamp, which is filled upon creation by the value of the `Timestamp` variable. This is a counter which is increased each time after initializing a stack structure and decreased after destroying the highest one.
The following instructions/places are modified for using the time stamp:

- the initialisation process of the query gets the very first entry
- the commands `try_me_else`, `retry_me_else`, `trust_me_else_fail`, `try`, `retry`, `trust`, `exec`, `ret`, `ret_const`, `ret_int`, `ret_float`, `allocate`, `deallocate`.

The rules for the commands for manipulating `Timestamp` are as follows:

1. `allocate`, `try_me_else`, and `try` create environments, respectively choice points, on top of the stack. So they copy the value of `Timestamp` into the time stamp slot and increase `Timestamp` by one.
2. `retry_me_else`, `retry` have to restore also `Timestamp`, which is set to the choice points time stamp plus one.
3. The rest of these commands remove a choice point/environment frame. the new timestamp is set to $max(timestamp(\texttt{E}), timestamp(\texttt{B})) + 1$, where `E` and `B` are the actual choice point/environment frame after the removal.

11.4.4 Access

Creating a stack frame is done via the function `StackAlloc`. There are two occasions where `StackAlloc` is called: for creating an environment or for creating a choice point. So, involved are the two of the registers `B` and `E` (and their auxiliary registers) for one of them refering to the current highest position on the stack and one of them being wanted to refer afterwards to the highest position. `StackAlloc` expects as arguments a pointer immediately after the highest stack frame, the `HST` where the topmost stack frame is contained (i.e. one of `CurrStack` and `CurrEStack`), the auxiliary `HST` register which may be updated, and the size of the stack frame to be allocated. It returns a reference to the allocated stack frame.

```
function StackAlloc(M : address, CS : address, CSN : address,
                    N : integer) : address

// arguments:
// M   : address immediately after current stack frame
// CS  : current highest stack HST
// CSN : stack HST which may be updated
// N   : size of stack frame
// S   : stack register to update

// test if enough space is left in the current HST memory:

 if CS->high.stack - M > N  then       // if yes, then everything is ok
  begin
   CSN = CS,
   return M;
  end;

// else, allocate new HST if needed and return reference to its low memory
 else
  begin
   if CS->next = 0 then
```

```
    if Stacksize + max(STACK_SIZE, N) <= STACK_MAX_SIZE then
     begin
      append_hstchunk(CS, max(STACK_SIZE, N), STACK);
      Stacksize = Stacksize + STACK_SIZE;
     end;

    else
     begin
      fail = true;
      return;
     end;

   CSN = CS->next;
   return CSN->low.stack;
  end;
end StackAlloc;
```

12 Auxiliary Items

12.1 Memory Management for Linked Lists

```
Memory_Area *init_mem_area(size_t type, size_t size, size_t enlargement);
void release_mem_area(Memory_Area **);
void *get_memory(Memory_Area *);
void release_memory(void *, Memory_Area *);
void clear_mem_area(Memory_Area *);
```

Because of the frequent use of linked lists during the assembling process, there exist - somewhat experimental - memory management routines which are taylored to quickly re-use memory allocated for linked lists. The idea is, not to allocate new system memory for each new entry in a linked list, but to allocate a whole memory block at once, mark locations in it as a new list entry and release all at once, if the linked list is no longer needed. The main data structure here is the `Memory_Area`:

Memory_Area		
$\longrightarrow$		
mem_blk	BYTE *	pointer to memory block
first_free	BYTE *	pointer to next element to be allocated
type	size_t	data type to be handled, "typed" by its size
size	size_t	number of possible entries
num_free	size_t	number of free entries
enlargement	size_t	extend `mem_blk` memory by this amount if exceeded

To handle a linked list via `Memory_Areas`, one has first to create the first `Memory_Area` with the function `init_mem_area`; its arguments are the type of entries in the list - which means its size -, the number of entries the `Memory_Area` can contain, and the size which an additional memory area will have, if the former number will be exceeded. For example,

```
init_mem_area(sizeof(Codeconstruct), 2000, 500);
```

initializes a `Memory_Area` for linked lists of type `Codeconstruct` (or some other of equivalent size, respectively) with a capacity of 2000 entries; if this is exceeded, additional `Memory_Areas` of 500 entries size will be appended.
When using the linked list, calls to `get_memory` will allocate a new element of the specified memory area. `get_memory` does it in the way, that it steps through all `Memory_Area` structures and returns the entry of the first free entry it finds. If it finds none, it calls `init_mem_area` to create a new `Memory_Area` and allocates its first entry.

release_memory searches for the Memory_Area where the specified list entry belongs to and releases it there.
If a linked list is no longer used, it is reset by the function clear_mem_area, which sets all Memory_Areas belonging to a memory area into an initial state.
To completely get rid of a memory area, one has to use release_mem_area.

13 Conclusions

This work gave an almost complete description of the RAWAM and its implementation. However, there remain some undiscussed and unfinished issues. The most important of these are the following: the unsufficient coupling of the RAWAM via the file system, the prototypical support of RelFun's module system, and the not yet completed adaption of the RAWAM to all of RelFun's features.
While the first point is just a matter of taste, the latter ones were delayed because of the fact that they would require both, the extension of the RAWAM and the reworking of the whole RFM compiler. For instance, the compiler currently flattens the modules, so, in fact, the module system in the RAWAM is only used for the distinction of predefined and user-defined WAM routines. Further open questions with respect to the support of a full module system are (1) how to implement a compilation/assembling scheme that allows backtracking across module contexts and (2) to give proper foundations of higher-order logic within a module system. These points are discussed in [Her95], [BLM94].
Summarizing, the RAWAM is well-suited and now well-enough described to be adapted to other logic programming languages; it is not only suited to work as an abstract machine, but its design also allows one further step: to reuse parts of it for generating C programs out of RelFun source, hence to provide native code for it (cf. [VR94]).

References

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, Cambridge, Massachusetts, 1991.

[AK95] Hassan Aït-Kaci. List of known bugs for the book: Hassan Aït-Kaci, "Warren's Abstract Machine: A Tutorial Reconstruction". http://www.isg.sfu.ca/~hak, 1995.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[BEH$^+$96] Harold Boley, Klaus Elsbernd, Hans-Guenther Hein, Thomas Krause, Markus Perling, Michael Sintek, and Werner Stein. RFM Manual: Compiling RELFUN into the Relational/Functional Machine. Document D-91-03, DFKI GmbH, July 1996. Third, Revised Edition.

[BLM94] Michele Bugliese, Evelina Lamma, and Paola Mello. Modularity in logic programming. *Journal of Logic Programming*, 19/20:443–502, 1994.

[Bol97] Harold Boley. A Relational-Functional Integration for Declarative Programming. In Christian Freksa, Matthias Jantzen, and Rüdiger Valk, editors, *Foundations of Computer Science : Potential - Theory - Cognition*, number 1337 in LNCS, pages 351–358. Springer-Verlag, Berlin, Heidelberg, 1997.

[GC96] Gopal Gupta and Mats Carlsson, editors. *The Journal of Logic Programming. Special Issue: High-Performance Implementations of Logic Programming Systems.* Elsevier Science Inc., 1996.

[GLLO85] John Gabriel, Tim Lindholm, E. L. Lusk, and R.A. Overbeek. A Tutorial on the Warren Abstract Machine for Computational Logic. Report ANL-84-84, Argonne National Laboratory, Argonne, Illinois 60439, June 1985.

[Hal95] Victoria Hall. Integration von Sorten als ausgezeichnete taxonomische Prädikate in eine relational-funktionale Sprache. Projektarbeit, March 1995.

[Her95] Michael Herfert. Deklarative statische und dynamische Softwaremodule. Master's thesis, Fachbereich Informatik, Universität Kaiserslautern, 1995.

[MW88] David Maier and David S. Warren. *Computing with Logic.* The Benjamin/Cummings Publishing Company, Inc., 1988.

[Nys85] Sven Olof Nystrøm. NyWam - A WAM Emulator Written in LISP, 1985.

[Per] Markus Perling. Rawam sources. http://www.dfki.uni-kl.de/~vega/relfun.html.

[Per97] Markus Perling. GeneTS: A Relational-Functional Genetic Algorithm for the Traveling Salesman Problem. Technical Report TM-97-01, DFKI GmbH, August 1997.

[Rus92] David M. Russinoff. A verified Prolog compiler for the Warren abstract machine. *Journal of Logic Programming,* 13(4):367–412, August 1992.

[Sin93] Michael Sintek. Indexing PROLOG Procedures into DAGs by Heuristic Classification. Technical Report TM-93-05, DFKI GmbH, 1993.

[Sin95] Michael Sintek. FLIP: Functional-plus-Logic Programming on an Integrated Platform. Master's thesis, Universität Kaiserslautern, 1995.

[VR94] Peter Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *The Journal of Logic Programming,* 19,20:385–441, 1994.

[War77] D. H. D. Warren. Compiling Predicate Logic Programs. D.A.I. Research Report, University of Edinbourgh, 1977.

[War83] David H. D. Warren. An abstract prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA, October 1983.

A Some Corrections

During the implementation of the RAWAM several errors contained in Aït-Kaci's original book [AK91] were found and corrected. We refrain here from describing these corrections in any detail, because there exists an official list of known bugs and their corrections. Since our independently found corrections completely agree with Aït-Kaci's, the current WAM version seems to be bug free.
Aït-Kaci's list can be found at his homepage at

`http://www.isg.sfu.ca/~hak/documents/wamerratum.txt`

B Overview: Supported WAM Instructions

B.1 Instructions as in Aït-Kaci's description of the WAM

The RAWAM instructions in this appendix all have counterparts in the WAM description [AK91], which are denoted in the second columns of the tables below.

B.1.1 Put instructions

RAWAM instructions	
`put_x_variable` X_n,X_m	`put_variable` X_n,A_i
`put_y_variable` Y_n,X_m	`put_variable` Y_n,A_i
`put_x_value` X_n,X_m	`put_value` V_n,A_i
`put_y_value` Y_n,X_m	`put_value` V_n,A_i
`put_unsafe_value` Y_n,X_m	`put_unsafe_value` Y_n,A_i
`put_structure` f,X_n	`put_structure` f,A_i
`put_list` X_n	`put_list` A_i
`put_constant` c,X_n	`put_constant` c,A_i
`put_nil` X_n	`put_constant` [],A_i
`put_int` i,X_n	`put_constant` i,A_i
`put_float` r,X_n	`put_constant` r,A_i

B.1.2 Get instructions

RAWAM instructions	
`get_x_variable` X_n,X_m	`get_variable` V_n,A_i
`get_y_variable` Y_n,X_m	`get_variable` V_n,A_i
`get_x_value` X_n,X_m	`get_value` V_n,A_i
`get_y_value` Y_n,X_m	`get_value` V_n,A_i
`get_structure` f,X_m	`get_structure` f,A_i
`get_list` X_n	`get_list` A_i
`get_constant` c,X_m	`get_constant` c,A_i
`get_nil` X_n	`get_constant` [],A_i
`get_int` i,X_m	`get_constant` i,A_i
`get_float` r,X_m	`get_constant` r,A_i

B.1.3 Unify instructions

RAWAM instructions	
unify_x_variable X_n	unify_variable V_n
unify_y_variable Y_n	unify_variable V_n
unify_x_value X_n	unify_value V_n
unify_y_value Y_n	unify_value V_n
unify_x_local_value X_n	unify_value V_n
unify_y_local_value Y_n	unify_value V_n
unify_constant c	unify_constant c
unify_nil X_n	unify_constant []
unify_int i	unify_constant i
unify_float r	unify_constant r
unify_void n	unify_void n

B.1.4 Control instructions

RAWAM instructions	
allocate	allocate
deallocate	deallocate
call P,N	call P,N
execute P	execute P
proceed	proceed

B.1.5 Choice instructions

RAWAM instructions	
try_me_else L,n	try_me_else L
retry_me_else L	retry_me_else L
trust_me_else_fail	trust_me
try L,n	try L
retry L	retry L
trust L	trust L

B.1.6 Indexing instructions

RAWAM instructions	
switch_on_term C,S,L,N,V	switch_on_term V,C,L,S
switch_on_constant N,T	switch_on_constant N,T
switch_on_structure N,T	switch_on_structure N,T

B.1.7 Cut instructions

RAWAM instructions	
neck_cut	neck_cut
get_level Y_n	get_level Y_n
cut Y_n	cut Y_n

B.2 Additional Instructions

B.2.1 Arithmetics Instructions

Instruction	Description
`add2`	adds contents of registers $X_1$and X_2
`addn n`	adds contents of registers $X_1$to X_n
`sub2`	subtracts contents of registers $X_1$and X_2
`subn n`	subtracts contents of registers $X_1$to X_n
`inc`	increases contents of register $X_1$by 1
`dec`	decreases contents of register $X_1$by 1
`mul2`	multiplies contents of registers $X_1$and X_2
`muln n`	multiplies contents of registers $X_1$to X_n
`div2`	divides contents of register $X_1$by X_2
`divn n`	divides contents of register $X_1$by $X_2$to X_n
`mod2`	X_1modulo X_2
`modn n`	X_1modulo X_2... modulo X_n
`random`	generates random value between $0 \ldots \{X_1\} - 1$ if X_1contains an integer, or between $0 \ldots \{X_1\}$ if X_1 contains a real number
`exp`	$e^{\{X_1\}}$
`expt`	$\{X_1\}^{\{X_2\}}$
`log`	$\ln\{X_1\}$
`sqrt`	$\sqrt{\{X_1\}}$
`abs`	$\mid \{X_1\} \mid$
`signum`	0 if $\{X_1\} = 0$, 1 if $\{X_1\} > 0$, -1 else
`sin`	$\sin\{X_1\}$
`cos`	$\cos\{X_1\}$
`tan`	$\tan\{X_1\}$
`asin`	$\arcsin\{X_1\}$
`acos`	$\arccos\{X_1\}$
`atan`	$\arctan\{X_1\}$
`pi`	π
`sinh`	$\sinh\{X_1\}$
`cosh`	$\cosh\{X_1\}$
`tanh`	$\tanh\{X_1\}$
`asinh`	$\text{arcsinh}\{X_1\}$
`acosh`	$\text{arccosh}\{X_1\}$
`atanh`	$\text{arctanh}\{X_1\}$
`lt2`	test if $\{X_1\} < \{X_2\}$
`ltn n`	test if $\{X_1\} < \cdots < \{X_n\}$
`le2`	test if $\{X_1\} \leq \{X_2\}$
`len n`	test if $\{X_1\} \leq \cdots \leq \{X_n\}$
`gt2`	test if $\{X_1\} > \{X_2\}$
`gtn n`	test if $\{X_1\} > \cdots > \{X_n\}$
`ge2`	test if $\{X_1\} \geq \{X_2\}$
`gen n`	test if $\{X_1\} \geq \cdots \geq \{X_n\}$
`eq2`	test if $\{X_1\} = \{X_2\}$
`eqn n`	test if $\{X_1\} = \cdots = \{X_n\}$
`ne2`	test if $\{X_1\} \neq \{X_2\}$
`nen n`	test if $\{X_1\} \neq \cdots \neq \{X_n\}$
`min n`	$\min \{X_1\} \ldots \{X_n\}$
`max n`	$\max \{X_1\} \ldots \{X_n\}$
`stringst`	test if $\{X_1\} < \{X_2\}$ and $\{X_1\}, \{X_2\}$ contain strings or constants
`stringse`	test if $\{X_1\} <= \{X_2\}$ and $\{X_1\}, \{X_2\}$ contain strings or constants
`stringgt`	test if $\{X_1\} > \{X_2\}$ and $\{X_1\}, \{X_2\}$ contain strings or constants
`stringge`	test if $\{X_1\} >= \{X_2\}$ and $\{X_1\}, \{X_2\}$ contain strings or constants
`stringeq`	test if $\{X_1\} = \{X_2\}$ and $\{X_1\}, \{X_2\}$ contain strings or constants
`stringneq`	test if $\{X_1\} \neq \{X_2\}$ and $\{X_1\}, \{X_2\}$ contain strings or constants

B.2.2 Miscellaneous Instructions

Indexing Instructions

Instruction	Description
`set_index_number` n	sets indexing register to X_n

Alternative Cut Instructions

Instruction	Corresponds to
`save_cut_pointer`	`get_level`
`gama_cut`	`cut`

Flow Control Instructions

Instruction	Description
`terminate`	terminates RAWAM execution of query
`failure`	terminates RAWAM execution and failures
`backtrack`	invokes backtracking

Special Commands

Instruction	Description
`type`	sort unification
`apply` t	higher order calls
`ll`	calls to lisp light
`trueatom`	true if X_1 contains an atom
`nontrueatom`	true if X_1 contains not an atom
`var`	true if X_1 contains a variable
`nonvar`	true if X_1 contains not a variable

B.2.3 Integrating Instructions

Instruction	Substitution
`exec` P	`deallocate` `execute` P
`ret`	`deallocate` `proceed`

`ret_const`	`put_constant` c,X_1
	`deallocate`
	`proceed`

`ret_int`	`put_int` i,X_1
	`deallocate`
	`proceed`

`ret_float`	`put_float` r,X_1
	`deallocate`
	`proceed`

`proceed_const`	`put_constant` c,X_1 `proceed`
`proceed_int`	`put_int` i,X_1 `proceed`
`proceed_float`	`put_float` r,X_1 `proceed`
`switch_on_term_n` n,C,S,L,N,V	`set_index_number` n `switch_on_term` C,S,L,N,V

C A Demo Script

```
rfi-p> exec rawam

relfun
rfi-p> %%%%%%%% rawam.bat
rfi-p> %%%%%%%% test facilities and speed-up of the RAWAM
rfi-p>
rfi-p> miser-level 3
rfi-p> inter
rfi-p> destroy
rfi-p> sp
rfi-p> timermode on
% timermode is on now.
rfi-p>
rfi-p>
rfi-p> %%% First a suite of the serialise-demo and
rfi-p> %%% comparison with the GWAM:
rfi-p>
rfi-p> %% pure functional version of serialise:
rfi-p>
rfi-p> consult funser
Reading file "./funser.rfp"
rfi-p> listing
serialise(L) :& assign(L,table(L,[])).
assign([],T) :& [].
assign([X|Rest],T) :& tup(assoc(X,T)|assign(Rest,T)).
assoc(X,[[X,L]|Rest]) !& L.
assoc(X,[[Y,L]|Rest]) :& assoc(X,Rest).
table([],T) :& T.
table([X|Rest],T) :- memb(X,T) !&table(Rest,T).
table([X|Rest],T) :& table(Rest,insert([X,1],T)).
memb(X,[[X,L]|Rest])!.
memb(X,[[Y,L]|Rest]) :- memb(X,Rest).
insert([X,L],[]) :& [[X,L]].
insert([X,L1],[[Y,L2]|Rest]) :-
      string<(X,Y) !&tup(tup(Y,1+(L2))|insert([X,L1],Rest)).
insert([X,L1],[[Y,L2]|Rest]) :&
         tup([Y,L2]|insert(tup(X,1+(L1)),Rest)).
loop(X,1) :- serialise(X).
loop(X,N) :- serialise(X) & loop(X,1-(N)).
rfi-p>
rfi-p> % try GWAM:
rfi-p>
rfi-p> emul
Collecting modules for the emulator:
sortbase workspace
rfe-p>
rfe-p> compile
rfe-p>
rfe-p> % first we look what it does:
rfe-p>
rfe-p> serialise([q,w,e,r,t,y,u,i,o,p,l,k,j,h,g,f,d,s,
a,z,x,c,v,b,n,m,a,s,d,f,r,e,w,q,z,x,c,v,b,g,t,y,h,n,m,
j,u,i,k,l,o,p,q,w,e,r,t,y,u,i,o,p,m,n,b,v,c,x,z,a,s,d,
f,g,h,j,k,l,q,a,z,x,c,v,b,n,m,s,d,f,g,h,j,k,l,w,e,r,t,
y,u,i,o,p])
[17,
 23,
 5,
 18,
 20,
 25,
 21,
 9,
 15,
 16,
 12,
 11,
 10,
```

```
8,
7,
6,
4,
19,
1,
26,
24,
3,
22,
2,
14,
13,
1,
19,
4,
6,
18,
5,
23,
17,
26,
24,
3,
22,
2,
7,
20,
25,
8,
14,
13,
10,
21,
9,
11,
12,
15,
16,
17,
23,
5,
18,
20,
25,
21,
9,
15,
16,
13,
14,
2,
22,
3,
24,
26,
1,
19,
4,
6,
7,
8,
10,
11,
12,
17,
1,
26,
 24,
 3,
 22,
 2,
 14,
 13,
 19,
 4,
 6,
 7,
 8,
 10,
 11,
 12,
 23,
 5,
 18,
 20,
 25,
 21,
 9,
 15,
 16]
% Internal run time:  2920 ticks (= 2.920000 sec)
rfe-p>
rfe-p> inter
rfi-p> pause()
true
% Internal run time:  0 ticks (= 0.000000 sec)
rfi-p> emul
Collecting modules for the emulator:
sortbase workspace
rfe-p>
rfe-p> % now we force GWAM to need some time:
rfe-p>
rfe-p> loop([q,w,e,r,t,y,u,i,o,p,l,k,j,h,g,f,d,s,
a,z,x,c,v,b,n,m,a,s,d,f,r,e,w,q,z,x,c,v,b,g,t,y,h,
n,m,j,u,i,k,l,o,p,q,w,e,r,t,y,u,i,o,p,m,n,b,v,c,x,
z,a,s,d,f,g,h,j,k,l,q,a,z,x,c,v,b,n,m,s,d,f,g,h,j,
k,l,w,e,r,t,y,u,i,o,p],10)
true
% Internal run time:  14390 ticks (= 14.390000 sec)
rfe-p>
rfe-p> % and again:
rfe-p>
rfe-p> ori
true
% Internal run time:  16000 ticks (= 16.000000 sec)
rfe-p>
rfe-p> inter
rfi-p> pause()
true
% Internal run time:  0 ticks (= 0.000000 sec)
rfi-p> emul
Collecting modules for the emulator:
sortbase workspace
rfe-p>
rfe-p> % now try RAWAM:
rfe-p>
rfe-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p> compile
rfc-p>
rfc-p> % test, how much time the compilation of
rfc-p> % the query takes:
rfc-p>
rfc-p> something_which_does_not_exist([q,w,e,r,t,y,u,i,
o,p,l,k,j,h,g,f,d,s,a,z,x,c,v,b,n,m,a,s,d,f,r,e,w,q,z,x,
c,v,b,g,t,y,h,n,m,j,u,i,k,l,o,p,q,w,e,r,t,y,u,i,o,p,m,n,
b,v,c,x,z,a,s,d,f,g,h,j,k,l,q,a,z,x,c,v,b,n,m,s,d,f,g,h,
j,k,l,w,e,r,t,y,u,i,o,p])
unknown
% Internal run time:  124 ticks (= 0.124000 sec)
rfc-p>
rfc-p> % subtract this time from the times below:
rfc-p>
rfc-p> serialise([q,w,e,r,t,y,u,i,o,p,l,k,j,h,g,f,d,s,
a,z,x,c,v,b,n,m,a,s,d,f,r,e,w,q,z,x,c,v,b,g,t,y,h,n,m,
j,u,i,k,l,o,p,q,w,e,r,t,y,u,i,o,p,m,n,b,v,c,x,z,a,s,d,
f,g,h,j,k,l,q,a,z,x,c,v,b,n,m,s,d,f,g,h,j,k,l,w,e,r,t,
y,u,i,o,p])
[17,
 23,
 5,
.
.
.         % some lines omitted here
.
.
 9,
 15,
 16]
% Internal run time:  234 ticks (= 0.234000 sec)
rfc-p>
rfc-p> loop([q,w,e,r,t,y,u,i,o,p,l,k,j,h,g,f,d,s,a,z,
x,c,v,b,n,m,a,s,d,f,r,e,w,q,z,x,c,v,b,g,t,y,h,n,m,j,u,
i,k,l,o,p,q,w,e,r,t,y,u,i,o,p,m,n,b,v,c,x,z,a,s,d,f,g,
h,j,k,l,q,a,z,x,c,v,b,n,m,s,d,f,g,h,j,k,l,w,e,r,t,y,u,
i,o,p],10)
true
% Internal run time:  584 ticks (= 0.584000 sec)
rfc-p>
rfc-p> ori
true
% Internal run time:  425 ticks (= 0.425000 sec)
rfc-p> ori
true
% Internal run time:  508 ticks (= 0.508000 sec)
rfc-p>
rfc-p> % Seen it?
rfc-p>
rfc-p> inter
rfi-p> pause()
true
% Internal run time:  0 ticks (= 0.000000 sec)
rfi-p>
rfi-p> destroy
rfi-p>
rfi-p> % This is a relational version of serialise:
rfi-p>
rfi-p> consult relser
Reading file "./relser.rfp"
rfi-p>
rfi-p> listing
apprel([],L,L).
apprel([H|R],L,[H|R1]) :- apprel(R,L,R1).
qsort[Cr]([],[]).
qsort[Cr]([X|Y],R) :-
        partition[Cr](X,Y,Sm,Gr),
        qsort[Cr](Sm,Sm-sorted),
        qsort[Cr](Gr,Gr-sorted),
        apprel(Sm-sorted,[X|Gr-sorted],R).
partition[Cr](X,[Y|Z],[Y|Smaller],Greater) :-
        Cr(Y,X),partition[Cr](X,Z,Smaller,Greater).
partition[Cr](X,[Y|Z],Smaller,[Y|Greater]) :-
        Cr(X,Y),partition[Cr](X,Z,Smaller,Greater).
partition[Cr](X,[X|Z],Smaller,Greater) :-
        partition[Cr](X,Z,Smaller,Greater).
partition[Cr](X,[],[],[]).
pairlists([X|L],[Y|R],[[X,Y]|P]) :- pairlists(L,R,P).
pairlists([],[],[]).
```

```
 24,
 3,
 22,
 2,
 14,
 13,
 19,
 4,
 6,
 7,
 8,
 10,
 11,
 12,
 23,
 5,
 18,
 20,
 25,
 21,
 9,
 15,
 16]
% Internal run time:  2920 ticks (= 2.920000 sec)
rfe-p>
rfe-p> inter
rfi-p> pause()
true
% Internal run time:  0 ticks (= 0.000000 sec)
rfi-p> emul
Collecting modules for the emulator:
sortbase workspace
rfe-p>
rfe-p> % now we force GWAM to need some time:
rfe-p>
rfe-p> loop([q,w,e,r,t,y,u,i,o,p,l,k,j,h,g,f,d,s,
a,z,x,c,v,b,n,m,a,s,d,f,r,e,w,q,z,x,c,v,b,g,t,y,h,
n,m,j,u,i,k,l,o,p,q,w,e,r,t,y,u,i,o,p,m,n,b,v,c,x,
z,a,s,d,f,g,h,j,k,l,q,a,z,x,c,v,b,n,m,s,d,f,g,h,j,
k,l,w,e,r,t,y,u,i,o,p],10)
true
% Internal run time:  14390 ticks (= 14.390000 sec)
rfe-p>
rfe-p> % and again:
rfe-p>
rfe-p> ori
true
% Internal run time:  16000 ticks (= 16.000000 sec)
rfe-p>
rfe-p> inter
rfi-p> pause()
true
% Internal run time:  0 ticks (= 0.000000 sec)
rfi-p> emul
Collecting modules for the emulator:
sortbase workspace
rfe-p>
rfe-p> % now try RAWAM:
rfe-p>
rfe-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p> compile
rfc-p>
rfc-p> % test, how much time the compilation of
rfc-p> % the query takes:
rfc-p>
rfc-p> something_which_does_not_exist([q,w,e,r,t,y,u,i,
o,p,l,k,j,h,g,f,d,s,a,z,x,c,v,b,n,m,a,s,d,f,r,e,w,q,z,x,
c,v,b,g,t,y,h,n,m,j,u,i,k,l,o,p,q,w,e,r,t,y,u,i,o,p,m,n,
b,v,c,x,z,a,s,d,f,g,h,j,k,l,q,a,z,x,c,v,b,n,m,s,d,f,g,h,
j,k,l,w,e,r,t,y,u,i,o,p])
unknown
% Internal run time:  124 ticks (= 0.124000 sec)
rfc-p>
rfc-p> % subtract this time from the times below:
rfc-p>
rfc-p> serialise([q,w,e,r,t,y,u,i,o,p,l,k,j,h,g,f,d,s,
a,z,x,c,v,b,n,m,a,s,d,f,r,e,w,q,z,x,c,v,b,g,t,y,h,n,m,
j,u,i,k,l,o,p,q,w,e,r,t,y,u,i,o,p,m,n,b,v,c,x,z,a,s,d,
f,g,h,j,k,l,q,a,z,x,c,v,b,n,m,s,d,f,g,h,j,k,l,w,e,r,t,
y,u,i,o,p])
[17,
 23,
 5,
.
.
.        % some lines omitted here
.
.
 9,
 15,
 16]
% Internal run time:  234 ticks (= 0.234000 sec)
rfc-p>
rfc-p> loop([q,w,e,r,t,y,u,i,o,p,l,k,j,h,g,f,d,s,a,z,
x,c,v,b,n,m,a,s,d,f,r,e,w,q,z,x,c,v,b,g,t,y,h,n,m,j,u,
i,k,l,o,p,q,w,e,r,t,y,u,i,o,p,m,n,b,v,c,x,z,a,s,d,f,g,
h,j,k,l,q,a,z,x,c,v,b,n,m,s,d,f,g,h,j,k,l,w,e,r,t,y,u,
i,o,p],10)
true
% Internal run time:  584 ticks (= 0.584000 sec)
rfc-p>
rfc-p> ori
true
% Internal run time:  425 ticks (= 0.425000 sec)
rfc-p> ori
true
% Internal run time:  508 ticks (= 0.508000 sec)
rfc-p>
rfc-p> % Seen it?
rfc-p>
rfc-p> inter
rfi-p> pause()
true
% Internal run time:  0 ticks (= 0.000000 sec)
rfi-p>
rfi-p> destroy
rfi-p>
rfi-p> % This is a relational version of serialise:
rfi-p>
rfi-p> consult relser
Reading file "./relser.rfp"
rfi-p>
rfi-p> listing
apprel([],L,L).
apprel([H|R],L,[H|R1]) :- apprel(R,L,R1).
qsort[Cr]([],[]).
qsort[Cr]([X|Y],R) :-
        partition[Cr](X,Y,Sm,Gr),
        qsort[Cr](Sm,Sm-sorted),
        qsort[Cr](Gr,Gr-sorted),
        apprel(Sm-sorted,[X|Gr-sorted],R).
partition[Cr](X,[Y|Z],[Y|Smaller],Greater) :-
        Cr(Y,X),partition[Cr](X,Z,Smaller,Greater).
partition[Cr](X,[Y|Z],Smaller,[Y|Greater]) :-
        Cr(X,Y),partition[Cr](X,Z,Smaller,Greater).
partition[Cr](X,[X|Z],Smaller,Greater) :-
        partition[Cr](X,Z,Smaller,Greater).
partition[Cr](X,[],[],[]).
pairlists([X|L],[Y|R],[[X,Y]|P]) :- pairlists(L,R,P).
pairlists([],[],[]).
```

```
numbered([[X,N]|R],N) :- numbered(R,1+(N)).
numbered([],N).
before([X1,Y1],[X2,Y2]) :- string<(X1,X2).
serialise(L,R) :- pairlists(L,R,P),qsort[before](P,N),
                  numbered(N,1).
loop(X,1) :- serialise(X,_).
loop(X,N) :- serialise(X,_) & loop(X,1-(N)).
rfi-p>
rfi-p> emul
Collecting modules for the emulator:
sortbase workspace
rfe-p> compile
rfe-p>
rfe-p> % it does the same:
rfe-p>
rfe-p> serialise([q,w,e,r,t,y,u,i,o,p,l,k,j,h,g,f,d,s,
a,z,x,c,v,b,n,m,a,s,d,f,r,e,w,q,z,x,c,v,b,g,t,y,h,n,m,
j,u,i,k,l,o,p,q,w,e,r,t,y,u,i,o,p,m,n,b,v,c,x,z,a,s,d,
f,g,h,j,k,l,q,a,z,x,c,v,b,n,m,s,d,f,g,h,j,k,l,w,e,r,t,
y,u,i,o,p],R)
true
R=[17,
   23,
.
.
.         % some lines omitted
.
.
   9,
   15,
   16]
% Internal run time:  710 ticks (= 0.710000 sec)
rfe-p>
rfe-p> inter
rfi-p> pause()
true
% Internal run time:  0 ticks (= 0.000000 sec)
rfi-p> emul
Collecting modules for the emulator:
sortbase workspace
rfe-p>
rfe-p> % do some work...
rfe-p>
rfe-p> loop([q,w,e,r,t,y,u,i,o,p,l,k,j,h,g,f,d,s,a,z,
x,c,v,b,n,m,a,s,d,f,r,e,w,q,z,x,c,v,b,g,t,y,h,n,m,j,u,
i,k,l,o,p,q,w,e,r,t,y,u,i,o,p,m,n,b,v,c,x,z,a,s,d,f,g,
h,j,k,l,q,a,z,x,c,v,b,n,m,s,d,f,g,h,j,k,l,w,e,r,t,y,u,
i,o,p],10)
true
% Internal run time:  5370 ticks (= 5.370000 sec)
rfe-p>
rfe-p> ori
true
% Internal run time:  5410 ticks (= 5.410000 sec)
rfe-p>
rfe-p> inter
rfi-p> pause()
true
% Internal run time:  10 ticks (= 0.010000 sec)
rfi-p>
rfi-p> % now we switch to the RAWAM:
rfi-p>
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p>
rfc-p> compile
rfc-p>
rfc-p> serialise([q,w,e,r,t,y,u,i,o,p,l,k,j,h,g,f,d,s,
a,z,x,c,v,b,n,m,a,s,d,f,r,e,w,q,z,x,c,v,b,g,t,y,h,n,m,
j,u,i,k,l,o,p,q,w,e,r,t,y,u,i,o,p,m,n,b,v,c,x,z,a
f,g,h,j,k,l,q,a,z,x,c,v,b,n,m,s,d,f,g,h,j,k,l,w,e
y,u,i,o,p],R)
true
R=[17,
   23,
.
.
.         % some lines omitted
.
.
   9,
   15,
   16]
% Internal run time:  257 ticks (= 0.257000 sec)
rfc-p>
rfc-p> inter
rfi-p> pause()
true
% Internal run time:  0 ticks (= 0.000000 sec)
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p>
rfc-p> loop([q,w,e,r,t,y,u,i,o,p,l,k,j,h,g,f,d,s,a
z,x,c,v,b,n,m,a,s,d,f,r,e,w,q,z,x,c,v,b,g,t,y,h,n,
j,u,i,k,l,o,p,q,w,e,r,t,y,u,i,o,p,m,n,b,v,c,x,z,a,
d,f,g,h,j,k,l,q,a,z,x,c,v,b,n,m,s,d,f,g,h,j,k,l,w,
r,t,y,u,i,o,p],10)
true
% Internal run time:  317 ticks (= 0.317000 sec)
rfc-p>
rfc-p> ori
true
% Internal run time:  261 ticks (= 0.261000 sec)
rfc-p> ori
true
% Internal run time:  265 ticks (= 0.265000 sec)
rfc-p>
rfc-p> inter
rfi-p> pause()
true
% Internal run time:  0 ticks (= 0.000000 sec)
rfi-p>
rfi-p> destroy
rfi-p>
rfi-p> consult mixser
Reading file "./mixser.rfp"
rfi-p>
rfi-p> emul
Collecting modules for the emulator:
sortbase workspace
rfe-p> compile
rfe-p>
rfe-p> serialise([q,w,e,r,t,y,u,i,o,p,l,k,j,h,g,f,
a,z,x,c,v,b,n,m,a,s,d,f,r,e,w,q,z,x,c,v,b,g,t,y,h,
j,u,i,k,l,o,p,q,w,e,r,t,y,u,i,o,p,m,n,b,v,c,x,z,a,
f,g,h,j,k,l,q,a,z,x,c,v,b,n,m,s,d,f,g,h,j,k,l,w,e,
y,u,i,o,p])
[17,
 23,
.
.
.
.
.
 9,
 15,
 16]
% Internal run time:  640 ticks (= 0.640000 sec)
rfe-p>
```

```
rfe-p> inter
rfi-p> pause()
true
% Internal run time:  0 ticks (= 0.000000 sec)
rfi-p> emul
Collecting modules for the emulator:
sortbase workspace
rfe-p>
rfe-p> % do some work...
rfe-p>
rfe-p> loop([q,w,e,r,t,y,u,i,o,p,l,k,j,h,g,f,d,s,a,z,
x,c,v,b,n,m,a,s,d,f,r,e,w,q,z,x,c,v,b,g,t,y,h,n,m,j,u,
i,k,l,o,p,q,w,e,r,t,y,u,i,o,p,m,n,b,v,c,x,z,a,s,d,f,g,
h,j,k,l,q,a,z,x,c,v,b,n,m,s,d,f,g,h,j,k,l,w,e,r,t,y,u,
i,o,p],10)
true
% Internal run time:  4900 ticks (= 4.900000 sec)
rfe-p>
rfe-p> ori
true
% Internal run time:  4690 ticks (= 4.690000 sec)
rfe-p>
rfe-p> inter
rfi-p> pause()
true
% Internal run time:  0 ticks (= 0.000000 sec)
rfi-p>
rfi-p> % now we switch to the RAWAM:
rfi-p>
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p>
rfc-p> compile
rfc-p>
rfc-p> serialise([q,w,e,r,t,y,u,i,o,p,l,k,j,h,g,f,d,s,
a,z,x,c,v,b,n,m,a,s,d,f,r,e,w,q,z,x,c,v,b,g,t,y,h,n,m,
,u,i,k,l,o,p,q,w,e,r,t,y,u,i,o,p,m,n,b,v,c,x,z,a,s,d,
f,g,h,j,k,l,q,a,z,x,c,v,b,n,m,s,d,f,g,h,j,k,l,w,e,r,t,
,u,i,o,p])
[17,
23,






9,
15,
16]
 Internal run time:  212 ticks (= 0.212000 sec)
fc-p>
fc-p> inter
fi-p> pause()
rue
 Internal run time:  0 ticks (= 0.000000 sec)
fi-p> emuc
ollecting modules for the emulator:
ortbase workspace
fc-p>
fc-p> loop([q,w,e,r,t,y,u,i,o,p,l,k,j,h,g,f,d,s,a,z,x,
,v,b,n,m,a,s,d,f,r,e,w,q,z,x,c,v,b,g,t,y,h,n,m,j,u,i,k,
,o,p,q,w,e,r,t,y,u,i,o,p,m,n,b,v,c,x,z,a,s,d,f,g,h,j,k,
,q,a,z,x,c,v,b,n,m,s,d,f,g,h,j,k,l,w,e,r,t,y,u,i,o,p],10)
rue
 Internal run time:  252 ticks (= 0.252000 sec)
fc-p>
fc-p> ori
rue
 Internal run time:  246 ticks (= 0.246000 sec)
fc-p> ori
```

```
true
% Internal run time:  246 ticks (= 0.246000 sec)
rfc-p>
rfc-p> inter
rfi-p> pause()
true
% Internal run time:  10 ticks (= 0.010000 sec)
rfi-p>
rfi-p> destroy
rfi-p> timermode off
% timermode is off now.
rfi-p>
rfi-p> %%% Now we demonstrate a genetic algorithm
rfi-p> %%% optimizing the TSP:
rfi-p>
rfi-p> consult ts
Reading file "./ts.rfp"
rfi-p>
rfi-p> % We have a set of 16 cities, arranged
rfi-p> % in a 4x4-array:
rfi-p>
rfi-p> listing plan2
plan2() :&
         [0.0,
          0.0,
          1.0,
          0.0,
          2.0,
          0.0,
          3.0,
          0.0,
          0.0,
          1.0,
          1.0,
          1.0,
          2.0,
          1.0,
          3.0,
          1.0,
          0.0,
          2.0,
          1.0,
          2.0,
          2.0,
          2.0,
          3.0,
          2.0,
          0.0,
          3.0,
          1.0,
          3.0,
          2.0,
          3.0,
          3.0,
          3.0].
rfi-p>
rfi-p> pause()
true
rfi-p>
rfi-p> % we access the algorithm via a testing clause
rfi-p>
rfi-p> listing test
test(Plan,Pop_size,Mut_rate,Cross_rate,Better_rate) :
       Map .= generate_distmap(Plan),
       Len .= /(length(Plan),2) &
       ts(init_pop(Pop_size,Len,Map,init_list(1,Len)),
        Map,
        Mut_rate,
        Cross_rate,
        Better_rate).
rfi-p>
```

```
rfi-p> pause()
true
rfi-p>
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p> compile
rfc-p>
rfc-p> % we use a population of 100 individuals,
rfc-p> % a mutation probability of 0.4,
rfc-p> % crossover probability of 0.1,
rfc-p> % and haploid genomes (1.0 = haploid,
rfc-p> % 0.0 = diploid)
rfc-p>
rfc-p> test(plan2(), 100, 0.4, 0.1, 1.0)
[24.064495,[16,14,13,15,3,7,10,9,5,6,11,2,1,4,8,12],32.93959]
rfc-p>
rfc-p> % we see the first generation
rfc-p> % output is the length of the shortest path within
rfc-p> % the population, the path itself wrt. to order given
rfc-p> % in the map, and the average path length within the
rfc-p> % population
rfc-p> % we will do some more optimization steps:
rfc-p>
rfc-p> inter
rfi-p> pause()
true
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p>
rfc-p> test(plan2(), 100, 0.4, 0.1, 1.0)
[25.877054,[3,16,12,7,6,10,9,15,11,5,13,14,8,4,2,1],32.610477]
rfc-p> more
[25.877054,[3,16,12,7,6,10,9,15,11,5,13,14,8,4,2,1],30.982065]
rfc-p> more
[25.877054,[3,16,12,7,6,10,9,15,11,5,13,14,8,4,2,1],29.482859]
rfc-p> more
[25.877054,[3,16,12,7,6,10,9,15,11,5,13,14,8,4,2,1],28.524343]
rfc-p> more
[25.877054,[3,16,12,7,6,10,9,15,11,5,13,14,8,4,2,1],27.671125]
rfc-p> more
[25.453745,[12,16,5,1,15,3,7,11,14,13,10,9,6,2,4,8],26.751875]
rfc-p> more
[25.320328,[10,11,15,6,5,7,2,1,9,14,12,16,8,4,3,13],26.09766]
rfc-p> more
[24.640985,[9,3,16,12,15,11,5,13,14,10,6,7,8,4,2,1],25.96922]
rfc-p> more
[22.714775,[10,11,12,14,9,1,2,7,3,4,8,16,15,6,5,13],25.59918]
rfc-p> more
[22.714775,[10,11,12,14,9,1,2,7,3,4,8,16,15,6,5,13],25.345667]
rfc-p> more
[22.472136,[3,7,6,10,9,15,11,5,13,14,16,12,8,4,2,1],25.245508]
rfc-p> more
[21.88635,[3,7,6,10,9,15,14,13,5,11,16,12,8,4,2,1],24.59224]
rfc-p> more
[21.88635,[3,7,6,10,9,15,14,13,5,11,16,12,8,4,2,1],24.03942]
rfc-p> more
[21.88635,[3,7,6,10,9,15,14,13,5,11,16,12,8,4,2,1],23.624447]
rfc-p> more
[21.88635,[3,7,6,10,9,15,14,13,5,11,16,12,8,4,2,1],23.086266]
rfc-p> more
[21.162277,[3,7,6,10,11,15,14,13,5,9,16,12,8,4,2,1],22.551796]
rfc-p> more
[20.650282,[2,7,6,10,9,11,5,13,14,15,16,12,8,4,3,1],22.344046]
rfc-p> more
[20.650282,[2,7,6,10,9,11,5,13,14,15,16,12,8,4,3,1],22.091103]
rfc-p> more
[20.650282,[2,7,6,10,9,11,5,13,14,15,16,12,8,4,3,1],21.942196]
rfc-p>
rfc-p> % the optimum value is 16 length units
```

```
rfc-p>
rfc-p>
rfc-p> %%% now we test extensively the types:
rfc-p>
rfc-p> exec rawamtypes.bat

relfun
rfc-p> % test types in the RAWAM
rfc-p> % this is a modification of typin.bat
rfc-p>
rfc-p> inter
rfi-p> sp
rfi-p> destroy
rfi-p> destroy-sortbase
rfi-p>
rfi-p> az drinks(mary,pina-colada).
rfi-p> az drinks(mary,vodka-lemon).
rfi-p> az drinks(mary,orange-flip).
rfi-p>
rfi-p> pause()
true
rfi-p>
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p> compile
rfc-p>
rfc-p> drinks(mary,What)
true
What=pina-colada
rfc-p> more
true
What=vodka-lemon
rfc-p> more
true
What=orange-flip
rfc-p> more
unknown
rfc-p>
rfc-p> inter
rfi-p> pause()
true
rfi-p>
rfi-p> destroy
rfi-p>
rfi-p> az drinks(mary,dom[pina-colada,
               vodka-lemon,orange-flip]).
rfi-p>
rfi-p> pause()
true
rfi-p>
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p> compile
rfc-p>
rfc-p> drinks(mary,What)
true
What=dom[pina-colada,vodka-lemon,orange-flip]
rfc-p>
rfc-p> drinks(mary,orange-flip)
true
rfc-p>
rfc-p> drinks(mary,whisky-sour)
unknown
rfc-p>
rfc-p> drinks(mary,dom[pina-colada,vodka-lemon,banana-flip
true
rfc-p>
rfc-p> drinks(mary,What),
       dom[pina-colada,vodka-lemon,banana-flip] .= What
```

```
true
What=dom[pina-colada,vodka-lemon]
rfc-p>
rfc-p> drinks(mary,What),
       dom[pina-colada,banana-flip] .= What
true
What=$pina-colada
rfc-p>
rfc-p> inter
rfi-p> pause()
true
rfi-p>
rfi-p> az drinks(john,exc[whisky-sour,vodka-lemon]).
rfi-p>
rfi-p> pause()
true
rfi-p>
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p> compile
rfc-p>
rfc-p> drinks(john,What)
true
What=exc[whisky-sour,vodka-lemon]
rfc-p>
rfc-p> drinks(john,orange-flip)
true
rfc-p>
rfc-p> drinks(john,whisky-sour)
unknown
rfc-p>
rfc-p> drinks(john,dom[pina-colada,orange-flip])
true
rfc-p> drinks(john,What), dom[pina-colada,orange-flip] .= What
true
What=dom[pina-colada,orange-flip]
rfc-p>
rfc-p> drinks(john,dom[pina-colada,whisky-sour])
true
rfc-p> drinks(john,What), dom[pina-colada,whisky-sour] .= What
true
What=$pina-colada
rfc-p>
rfc-p> drinks(john,dom[whisky-sour,vodka-lemon])
unknown
rfc-p> drinks(john,What), dom[whisky-sour,vodka-lemon] .= What
unknown
rfc-p>
rfc-p> drinks(mary,What), drinks(john,What)
true
What=dom[pina-colada,orange-flip]
rfc-p>
rfc-p> inter
rfi-p> destroy
rfi-p>
rfi-p> pause()
true
rfi-p>
rfi-p> mcd sortbase
Module:  sortbase
Context:
rfi-p>
rfi-p> az subsumes(cocktail,lightmix).
rfi-p> az subsumes(cocktail,heavymix).
rfi-p>
rfi-p> az lightmix(pina-colada).
rfi-p> az lightmix(vodka-lemon).
rfi-p> az lightmix(orange-flip).
rfi-p>
rfi-p> az heavymix(whisky-sour).
rfi-p> az heavymix("bloody-mary, strong").
rfi-p>
rfi-p> mcd
Module:  workspace
Context:
rfi-p> compile-sortbase
rfi-p>
rfi-p> pause()
true
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p> compile
rfc-p>
rfc-p> $lightmix .= $cocktail
$lightmix
rfc-p> $heavymix .= $cocktail
$heavymix
rfc-p> $lightmix .= $heavymix
unknown
rfc-p>
rfc-p> inter
rfi-p>
rfi-p> az drinks(mary,$lightmix).
rfi-p>
rfi-p> pause()
true
rfi-p>
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p> compile
rfc-p>
rfc-p> drinks(mary,What)
true
What=$lightmix
rfc-p>
rfc-p> drinks(mary,orange-flip)
true
rfc-p>
rfc-p> drinks(mary,whisky-sour)
unknown
rfc-p>
rfc-p> drinks(mary,dom[pina-colada,vodka-lemon,banana-flip
true
rfc-p>
rfc-p> drinks(mary,What),
       dom[pina-colada,vodka-lemon,banana-flip] .= What
true
What=dom[pina-colada,vodka-lemon]
rfc-p>
rfc-p> inter
rfi-p>
rfi-p> az drinks(fred,vodka-lemon).
rfi-p>
rfi-p> pause()
true
rfi-p>
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p> compile
rfc-p>
rfc-p> drinks(fred,$lightmix)
true
rfc-p>
rfc-p> inter
rfi-p> pause()
true
rfi-p>
rfi-p> az drinks(sue,"Barbara's special green-mix").
```

```
rfi-p>
rfi-p> pause()
true
rfi-p>
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p> compile
rfc-p>
rfc-p> drinks(sue,$atom)
true
rfc-p> drinks(sue,$numberp)
unknown
rfc-p> drinks(sue,$stringp)
true
rfc-p>
rfc-p> inter
rfi-p> pause()
true
rfi-p>
rfi-p> az drinks(jack,dom["Juan's drink",honey-liqueur,
                         "Boston ward 8"]).
rfi-p>
rfi-p> pause()
true
rfi-p>
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p> compile
rfc-p>
rfc-p> drinks(jack,What), $stringp .= What
true
What=dom["Juan's drink","Boston ward 8"]
rfc-p>
rfc-p> $stringp .= $heavymix
"bloody-mary, strong"
rfc-p> $symbolp .= $heavymix
dom[whisky-sour,"bloody-mary, strong"]
rfc-p> $numberp .= $heavymix
unknown
rfc-p>
rfc-p> inter
rfi-p> pause()
true
rfi-p>
rfi-p> az orders(laura,whisky-sour).
rfi-p> az drinks(peter,bnd[M,dom[whisky-sour,
                 "bloody-mary, strong"]]) :- orders(S,M).
rfi-p>
rfi-p> pause()
true
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p> compile
rfc-p>
rfc-p> drinks(peter,What)
true
What=whisky-sour
rfc-p>
rfc-p> inter
rfi-p> pause()
true
rfi-p>
rfi-p> rx drinks(peter,bnd[M,dom[whisky-sour,
                "bloody-mary, strong"]]) :- orders(S,M).
rfi-p>
rfi-p> az drinks(steve,bnd[M,exc[whisky-sour,vodka-lemon]])
          :- orders(S,M).
rfi-p>
```

```
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p> compile
rfc-p>
rfc-p> drinks(steve,What)
unknown
rfc-p>
rfc-p> inter
rfi-p> pause()
true
rfi-p>
rfi-p> az drinks(peter,bnd[M,$heavymix]) :- orders(S,M).
rfi-p>
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p> compile
rfc-p>
rfc-p> drinks(peter,What)
true
What=whisky-sour
rfc-p>
rfc-p> inter
rfi-p> pause()
true
rfi-p>
rfi-p> az drinks(adrian,bnd[M,$atom]) :- orders(S,M).
rfi-p>
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p> compile
rfc-p>
rfc-p> drinks(adrian,What)
true
What=whisky-sour
rfc-p>
rfc-p> inter
rfi-p>
rfi-p> mcd sortbase
Module:  sortbase
Context:
rfi-p>
rfi-p> az person(steve).
rfi-p> az person(john).
rfi-p> az person(mary).
rfi-p>
rfi-p> mcd
Module:  workspace
Context:
rfi-p> compile-sortbase
rfi-p>
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p> compile
rfc-p>
rfc-p> X .= [_], [mary] .= X
true
X=[mary]
rfc-p> X .= [_], [mary] .= X, [john] .= X
unknown
rfc-p>
rfc-p> X .= [dom[john,mary]], [mary] .= X
true
X=[mary]
rfc-p> X .= [dom[john,mary]], [mary] .= X, [john] .= X
unknown
rfc-p>
rfc-p> X .= [exc[fred]], [mary] .= X
```

```
true
X=[mary]
rfc-p> X .= [exc[fred]], [mary] .= X, [john] .= X
unknown
rfc-p>
rfc-p> X .= [$person], [mary] .= X
true
X=[mary]
rfc-p> X .= [$person], [mary] .= X, [john] .= X
unknown
rfc-p>
rfc-p> X .= [$symbolp], [mary] .= X
true
X=[mary]
rfc-p> X .= [$symbolp], [mary] .= X, [john] .= X
unknown
rfc-p>
rfc-p> inter
rfi-p> pause()
true
rfi-p>
rfi-p> az drinks(X,soft-drink).
rfi-p>
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p> compile
rfc-p>
```

```
rfc-p> drinks(steve,What)
true
What=soft-drink
rfc-p> drinks(tweety,What)
true
What=soft-drink
rfc-p>
rfc-p> inter
rfi-p> pause()
true
rfi-p>
rfi-p> style lisp
rfi-l>
rfi-l> a0 (sg (drinks $person _something))
rfi-l>
rfi-l> style prolog
rfi-p>
rfi-p> emuc
Collecting modules for the emulator:
sortbase workspace
rfc-p> compile
rfc-p>
rfc-p> drinks(steve,What)
true
What=soft-drink
rfc-p> drinks(tweety,What)
unknown
rfc-p>
```