Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-91-03

RFM Manual: Compiling RELFUN into the Relational/Functional Machine

Harold Boley, Klaus Elsbernd, Hans-Günther Hein,
Thomas Krause, Markus Perling, Michael Sintek, Werner Stein

Third, Revised Edition

July 1996

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Deutsches Forschungszentrum
für
Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, Sema Group, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry of Education, Science, Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct systems with technical knowledge and common sense which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- ☐ Intelligent Engineering Systems
- ☐ Intelligent User Interfaces
- ☐ Computer Linguistics
- ☐ Programming Systems
- ☐ Deduction and Multiagent Systems
- ☐ Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Dr. Dr. D. Ruland
Director

RFM Manual:
Compiling RELFUN into the Relational/Functional Machine

(Third, Revised Edition)

Harold Boley, Klaus Elsbernd, Hans-Günther Hein, Thomas Krause,
Markus Perling, Michael Sintek, Werner Stein

DFKI-D-91-03

This work has been supported by a grant from The Federal Ministry for Research and Technology (FKZ ITW-8902 C4).

RFM Manual: Compiling **RELFUN** into the Relational/Functional Machine

Harold Boley, Klaus Elsbernd, Hans-Günther Hein,
Thomas Krause, Markus Perling, Michael Sintek, Werner Stein

DFKI
Universität Kaiserslautern
Erwin-Schrödinger-Straße
67663 Kaiserslautern
Germany

Third, Revised Edition

July 1996

Abstract

The compilation of **RELFUN** programs consists of two main stages, horizontal transformations and vertical translations. The horizontal transformer performs both source-to-source steps into a subset of **RELFUN** and source-to-intermediate steps into a **RELFUN**-like language. The vertical translator is also divided into two phases, the classifier and the code generator. The classifier produces a declarative clause language; the code generator optimizes target code for underlying WAM emulators. These parts can be used *incrementally-individually*, as a relational/functional compilation laboratory, or *batch-composed*, as a complete **RELFUN** compiler. All intermediate steps employ explicit declarative representations, which can be displayed via **RELFUN**'s user interface. The compiler is implemented in a subset of COMMON LISP; one emulator runs in COMMON LISP, the other in ANSI C.

Contents

1 Introduction

This work describes the compilation and execution environment of the Relational/Functional Machine (**RFM**). The **RFM** is a LISP/C-based implementation of **RELFUN** [Bol92] and consists of an interpreter, a multi-pass compiler, and two emulators.

The compilation of **RELFUN** programs consists of two main stages, horizontal transformations and vertical translations. The horizontal transformer is divided into several steps, whose target is mainly a simpler subset of **RELFUN**, but for advanced features can also be extended representations. The ensuing vertical translator is divided into two stages, the classifier and the code generator. The classifier transforms **RELFUN** source clauses to so-called "Classified Clauses"; from these WAM-annotated clauses the code generator can almost 'read off' the WAM code (see below).

All compilation steps can be used separately, as a compilation laboratory, as well as batch-composed, as a complete **RELFUN** compiler. Of course, various groups of these steps could be joined into single steps for optimizing *compilation* time. But organizing the compiler into such steps enhances its modularity and readability, which helps in the development of optimizations of *execution* time, our main concern.

Both emulators are extensions of the WAM (Warren Abstract Machine). The first emulator is called **GWAM** (Generalized WAM [Sin95]), the successor to the NyWAM [Hei89], which originated from Nystrøm's WAM [Nys]. The **GWAM** is built in COMMON LISP on a general implementation platform, the **GAMA** (General Abstract Machine [Sin95]), which contains a debugger, an assembler, and a loader. The second emulator is called **RAWAM** (Relfun Adapted WAM), more based on [AK91], and built in ANSI C [Per96].

It is assumed that the reader be somewhat familiar with **RELFUN** (see [BAE$^+$96]), and with WAM architectures ([War83], [AK91], [VR94]). For further information about the **RFM** see [Bol92] [Kra90], [Hei89], [Els90].

The user interface of the **RFM** is described in section 2. The horizontal transformations are the subject of section 3.1. Sections 4 and 5 treat the classifier and code generator for vertical translations; sections 6 and 7, the **GAMA** and the embedded **GWAM** emulator. The last section contains an example dialog that will show some aspects of the compiler/emulator system 'live'.

2 The user interface

The user interface provides several commands each of which represents a separate compilation step. The commands are hierarchically structured and top-down ordered as depicted by the indentation tree below: Each node can be called individually; inner nodes perform groups of compilation steps so that the root is the complete compiler.

The command hierarchy:

```
compile
   horizon
      extron
        undeclare
        untype
        unmacro
        unor
        unlambda
        hitrans
        uncomma
      bastron
        untup
        flatter
        passtup
        deanon
        normalize
        footen
   verti
      classify
      codegen
```

The given order reflects the order the commands are executed during **RELFUN** compilation.

2.1 The user interface for layered compilation

The compilation of **RELFUN** clauses into WAM code is done in several steps; the user interface enables to execute each step or groups of compatible steps separately.

The complete compiler is invoked by the `compile` command; it can be called with an extra argument for compiling a single procedure, thus allowing procedure-based incremental compilation. The `compile` command is divided into two stages, the precompilation (horizontal transformations) and the proper compilation (vertical translations). The horizontal transformations are performed by the `horizon` command, the vertical translations by the `verti` command.

`horizon` is itself divided into two parts, `extron` and `bastron`. The `extron` transformations `undeclare`, `untype`, `unmacro`, and `unor` map into extended constructs, in particular `lambda` expressions, which are then further transformed by `unlamda` and `uncomma` into a **RELFUN** subset (these are described in section 3.1). The `bastron` transformations convert these reduced **RELFUN** clauses into an even smaller subset that is ready for the vertical transformations. E.g. at the time of the `verti` command all `tup`s will have been transformed into `cns` structures via the `untup` command; it is also assumed that only flattened clauses are in the database, which is performed by the `flatter` command (the `bastron` transformations are described in section 3.2).

`verti` consists of two phases, the classifier and the code generator. Like in `horizon` these phases can also be called explicitely by typing `classify` and `codegen`. The `classify` command collects all clauses starting with the same name and arity, and groups them together on the property list of the symbol determined by the procedure name, using the tag `clauses`. This is necessary because the basic entity in the WAM is a group of clauses with the same name and arity, called a 'procedure'. After this, the classified clauses are generated and stored in a global variable called `*classified-database*`. The `codegen` command reads the contents of `*classified-database*` and produces **GWAM** code from it.

It is possible to pretty print the classified clauses by typing `listclass` and the code with the `listcode` command.

2.2 The user interface and the GWAM

The user interface has four prompts[1]: "`rfi-p>`" or "`rfi-l>`" is displayed when the queries are sent to the interpreter and its database, while "`rfe-p>` " and "`rfe-l>` " show that the query, which is a valued conjunction of $n \geq 1$ literals, will be emulated after compilation. The suffix of the prompt is "`-p>`" or "`-l>`", respectively, when the system is running in PROLOG or LISP style (see [Her92]). The code obtained is stored under the name `main`, the data structures for the variables in the query are created and their names and locations are memorized to get the variable names when the goal succeeds. Finally the emulator is called, producing failures or returned values with possible variable bindings. When a goal succeeds, the results are printed; backtracking is invoked if the user's next input is `more` so that the next solution may be computed. When **spy** is enabled, the query's compilation is shown and the **GWAM** is set into debugging (interactive or non-interactive single-step) mode. With **nospy** this feature is turned off.

3 The transformers

The transformers behind the `horizon` command 'horizontally' map RELFUN source programs to source programs that are either still in RELFUN (subsection 3.2) or in an extended high-level language (subsection 3.1). Both kinds of transformers lay the ground for later compilers 'vertically' proceeding into the WAM.

While some of the transformer steps can be performed independently from the other ones, many require previous transformers as a precondition for obtaining their effect (all transformers just deliver a database unchanged if they are inapplicable, either because their pretransformations are still missing or their fixpoint is reached). While the order shown in the command hierarchy of section 2 need not be obeyed totally, in the following we use it as the canonical order rather than indicating more detailed dependencies.

---

[1] There is one additional prompt, "`ll>`", for LISP *light* (see [Sin95])

3.1 The extron transformers

These transformations principally reduce language extensions to an unextended kernel. The sequence of these transformations, shown in the command hierarchy, is reflected by the subsection ordering.

3.1.1 undeclare

undeclare handles two different kinds of declarations: signature declarations (sg clauses) and declare facts which are used for various declaration types.

undeclare performs the following three steps:

1. transform operators with sg definitions
2. evaluate declare facts
3. remove declare facts

Transforming sg definitions

The transformation of operators which contain sg definitions is shown in the following example, a definition of Fibonacci numbers working on both ordinary integers and their successor representation.

Applying undeclare to this operator transforms each sg definition into an ordinary (ft) clause which calls an operator fib.n ($n \in \{1,2,3\}$). The definitions of fib.n are obtained simply by renaming the original fib clauses, using fib.1 for the first sg-block, fib.2 for the second, and fib.3 for the third.

```
sg fib($integerp).

fib(0) :- & 1.
fib(1) :- & 1.
fib(N) :- & +(fib(-(N,1)),
              fib(-(N,2))).

sg fib(null).

fib(null) :- & s[null].

sg fib(s[X]).

fib(s[null]) :- & s[null].
fib(N) :-
         sub1(N,Nm1),
         sub1(Nm1,Nm2),
         R1 is fib(Nm1),
         R2 is fib(Nm2),
         plus(R1,R2,R) &
         R.
```

```
fib(bnd[Arg#1,$integerp]) :- &
         fib.1(Arg#1).
fib.1(0) :- & 1.
fib.1(1) :- & 1.
fib.1(N) :- & +(fib(-(N,1)),
                fib(-(N,2))).

fib(bnd[Arg#1,null]) :- &
         fib.2(Arg#1).
fib.2(null) :- & s[null].

fib(bnd[Arg#1,s[X]]) :- &
         fib.3(Arg#1).
fib.3(s[null]) :- & s[null].
fib.3(N) :-
         sub1(N,Nm1),
         sub1(Nm1,Nm2),
         R1 is fib(Nm1),
         R2 is fib(Nm2),
         plus(R1,R2,R) &
         R.
```

```
(sg (fib $integerp))            (ft (fib (bnd _arg#1 $integerp))
                                    (fib.1 _arg#1) )
(ft (fib 0) 1 )                 (ft (fib.1 0) 1)
(ft (fib 1) 1 )                 (ft (fib.1 1) 1)
(ft (fib _n)                    (ft (fib.1 _n)
    (+ (fib (- _n 1))               (+ (fib (- _n 1))
       (fib (- _n 2)) ) )              (fib (- _n 2)) ) )

(sg (fib null))                 (ft (fib (bnd _arg#1 null))
                                    (fib.2 _arg#1) )
(ft (fib null) `(s null))       (ft (fib.2 null) `(s null) )

(sg (fib (s _x)))               (ft (fib (bnd _arg#1 (s _x)))
                                    (fib.3 _arg#1) )
(ft (fib (s null))              (ft (fib.3 (s null))
    `(s null))                      `(s null) )
(ft (fib _n)                    (ft (fib.3 _n)
    (sub1 _n _nm1)                  (sub1 _n _nm1)
    (sub1 _nm1 _nm2)                (sub1 _nm1 _nm2)
    (is _r1 (fib _nm1))             (is _r1 (fib _nm1))
    (is _r2 (fib _nm2))             (is _r2 (fib _nm2))
    (plus _r1 _r2 _r)               (plus _r1 _r2 _r)
    _r )                            _r )
```

Evaluating declare facts

The general form of a declare fact is as follows:

`declare(`*tag*`[`arg_1,..., arg_n`], ...).`

where *tag*`[`arg_1,...,arg_n`]` can be, amongst some others[2], one of

- `info[`*term*,...`]` — print *term*,... at compile time
- `tupstruct[`*atom*,...`]` — declare *atom*,... to be structure/operator names that must be handled like lists to allow them to be used with varying arity ("|"-operator)
- `macro[`*name, functional-object*`]` — declare a macro to be transformed by unmacro (since *functional-object* is a COMMON LISP functional object, using the macro feature is not encouraged)

3.1.2 untype

untype transforms types[3], i.e. domains (dom-terms), exclusions (exc-terms),

---

[2] `defun` to define COMMON LISP functions used by `macro`, `proto-class` and `indi-class` for defining ORF classes, `ll` and `llp` to define LISP *light* functions and predicates accessible by RELFUN, and `mode` and `dfmode` for mode declarations currently used for the transformation of RELFUN operators into LISP *light* functions.

[3] In addition to types, `untype` also handles ORF clauses which are not described in this paper.

and sorts ("\$"-terms) into active calls of the type/type1[4] builtin (which is only available in compiled RELFUN). Furthermore, expressions of the form *expr* : *type* and bnd[*expr*$_1$, *expr*$_2$] are handled by transforming them into is-calls or type-calls.

The meaning of type(*term, tterm*), where *tterm* is either a dom-term, an exc-term, or an atom (denoting the name of a sort), is: if *term* is a variable, type it with *tterm* (i.e., fill the type slot of the GWAM representation of variables, ref-cells, with *tterm*), otherwise check if *term* is of type *tterm*.

The following examples show some of the cases covered by untype:[5]

```
p() :- q($integerp).              | p() :- q(type1(integerp)).
p(X) :- X is dom[1,2,3].          | p(X) :- X is type1(dom[1,2,3]).
p(X) :- X : dom[1,2,3].           | p(X) :- X is type1(dom[1,2,3]).
p(exc[1,2,3]).                    | p(type1(exc[1,2,3])).
p(X : $realp) :- q(X).            | p(type(X,realp)) :- q(X).
p(bnd[X,$realp]) :- q(X).         | p(type(X,realp)) :- q(X).
```

```
(hn (p) (q $integerp))              | (hn (p) (q (type1 integerp)))
(hn (p _x) (is _x '(dom 1 2 3)))    | (hn (p _x)
                                    |     (is _x `,(type1 '(dom 1 2 3))) )
(hn (p _x) (_x : '(dom 1 2 3)))     | (hn (p _x)
                                    |     (is _x `,(type1 '(dom 1 2 3))) )
(hn (p (exc 1 2 3)))                | (hn (p ,(type1 '(exc 1 2 3))))
(hn (p (_x : $realp)) (q _x))       | (hn (p ,(type _x realp)) (q _x))
(hn (p (bnd _x $realp)) (q _x))     | (hn (p ,(type _x realp)) (q _x))
```

3.1.3 unmacro

unmacro is a transformation tool that handles various predefined as well as user-defined macros.

User-defined macros are declared with declare facts (see section 3.1.1). Since the syntactic transformation performed by these macros is defined via COMMON LISP functional objects, using them is not encouraged and thus not further described in this paper.

The following macros are predefined:

- progn simply denotes an inline conjunction of expressions, returning the value of the last one (analogously to LISP); unmcaro transforms it into a simple lambda application, which will be removed by hitrans (see section 3.1.6): $\dfrac{\texttt{(progn } p_1 \ldots p_n\texttt{)}}{\texttt{((lambda () } p'_1 \ldots p'_n\texttt{))}}$
- let creates a context with local (v_i) and auxiliary variables (a_i) in which some premises (p_i) are evaluated:

---

[4] type1(*tterm*) is the short form of type(_, *tterm*) and is expanded by unmacro.

[5] In our current implementation, RELFUN does not handle ","-expressions (see section 3.1.7) when using PROLOG syntax. In this paper, expressions like `(s _x ,(p _y))` and (hn (q _x ,(p _y))) are shown as s[X, p(Y)] and q(X, p(Y)). in PROLOG-like syntax.

`(let ((`v_1 e_1`) ... (`v_n e_n`)` a_1 ... a_m`)` p_1 ... p_o`)`
Its meaning is identical to that in COMMON LISP; it is, analogously to progn, translated into `lambda` expressions.

- `let*`, just like `let`, creates a local context, but does not evaluate the expressions e_i in parallel but sequentially (just like its COMMON LISP counterpart), thus allowing any v_i to access any v_j with $j \leq i$.
- `new-once` is the new version[6] of once used in compiled RELFUN which allows multiple expressions, returning the value of the last one, which are enclosed in a single `lambda` expression:

$$\frac{\texttt{(new-once } p_1 \ \ldots \ p_n\texttt{)}}{\texttt{(new-once (lambda () } p'_1 \ \ldots \ p'_n\texttt{))}}$$

- `naf` is handled analogously to `new-once`:

$$\frac{\texttt{(naf } p_1 \ \ldots \ p_n\texttt{)}}{\texttt{(naf* (lambda () } p'_1 \ \ldots \ p'_n\texttt{))}}$$

- `tupof` is handled analogously to `new-once`:

$$\frac{\texttt{(tupof } p_1 \ \ldots \ p_n\texttt{)}}{\texttt{(tupof* (lambda () } p'_1 \ \ldots \ p'_n\texttt{))}}$$

- "`!`" is transformed into an active call, `(cut)`, in order to simplify the vertical compiler:

$$\frac{\texttt{!}}{\texttt{(cut)}}$$

- `type1` is expanded to `type` with an anonymous variable:

$$\frac{\texttt{(type1 } t\texttt{)}}{\texttt{(type id } t\texttt{)}}$$

The following examples show how `let` and `let*` are transformed into `lambda` applications. Since we did not yet develop a PROLOG-like syntax for these constructs, only the LISP-like syntax is shown.

```
(hn (p _x _y)
  (is _y
      (let ((_a 1) (_b 2) (_y 3) _ab)
           (p _a _b _ab)
           (+ _ab _x _y))))


(ft (q _x _y)
  (let* ((_a (+ _x _y))
         (_b (* _a 5)))
        (/ _a _b)))
```

```
(hn (p _x _y)
    (is _y
     ((lambda (_a _b _y &aux _ab)
        (p _a _b _ab)
        (+ _ab _x _y) )
      1 2 3 ) ) )


(ft (q _x _y)
    ((lambda (&aux _a _b)
       (is _rb1 (+ _x _y))
       (is _a _rb1)
       (is _rb2 (* _a 5))
       (is _b _rb2)
       (/ _a _b) )) )
```

---

[6] The name `new-once` is used for historical reasons, as well as its transformation into another `new-once` and not into a `new-once*`.

3.1.4 unor

unor transforms inline disjunctions into corresponding, argument-less **lambda** applications, which are removed by **unlambda** using separate clauses (see section 3.1.5).

```
p(X,Y,Z) :-
  or( Z is +(X,Y), Z is *(X,Y) ).

(hn (p _x _y _z)
    (or
     (is _z (+ _x _y))
     (is _z (* _x _y)) ) )
```

```
(hn (p _x _y _z)
    ((lambda ()
      (or
       (is _z (+ _x _y))
       (is _z (* _x _y)) ) )) )
```

3.1.5 unlambda

unlambda transforms **lambda** expressions that cannot be expanded inline[7], i.e. additional clauses are generated:

- if a lambda expression is used as a value (as in **(is _l (lambda (_a _b) ...))**, a single clause containing the **lambda** literals is generated;
- if a lambda expression contains an or as its only literal (as introduced by unor), a clause is generated for each of the or literals.

In both cases, the **lambda** expression is replaced by a structure ‘(lambda$n$ f_1 ... f_m), where lambda$n$ is a new symbol created by **gentemp** and f_1 ... f_m are the free variables occurring in the lambda expression (for $m = 0$, instead of ‘(lambda$n$), only a new constant lambda$n$ is generated).

```
(hn (p _x _y)
  (is _c 5)
  (is _l (lambda (_a _b)
           (+ _a _b _c)))
  (_l _x _y))
```

```
(hn (p _x _y)
    (is _c 5)
    (is _l ‘(lambda1 _c))
    (_l _x _y) )
(ft ((lambda1 _c) _a _b)
    (+ _a _b _c) )
```

```
(hn (p _x _y _z)
    ((lambda ()
      (or
       (is _z (+ _x _y))
       (is _z (* _x _y)) ) )) )
```

```
(hn (p _x _y _z)
    (‘(lambda2 _z _x _y)) )
(ft ((lambda2 _z _x _y))
    (is _z (+ _x _y)) )
(ft ((lambda2 _z _x _y))
    (is _z (* _x _y)) )
```

3.1.6 hitrans

hitrans reduces higher-order expressions to apply calls. Furthermore, structures in functor positions are flattened.

---

[7] Inline expandable lambda expressions are transformed by uncomma (see section 3.1.7).

```
sorted[Comp]([A,B|R]) :-
  Comp(A,B),
  sorted[Comp]([B|R]).
```

```
sorted(Comp,[A,B|R]) :-
  apply(Comp,tup(A,B),user),
  sorted(Comp,[B|R]).
```

```
(hn ((sorted _comp)
     (tup _a _b | _r))
    (_comp _a _b)
    (`(sorted _comp)
      `(tup _b | _r)) )
```

```
(hn (sorted _comp (tup _a _b | _r))
    (apply _comp (tup _a _b) user)
    (sorted _comp `(tup _b | _r)) )
```

3.1.7 uncomma

uncomma transforms ","-expressions, which are used to activate expressions inside of structures, and inline expandable `lambda` applications.

```
(ft (p _x _y) `(s _x _y ,(+ _x _y)))



(hn (p _x _y)
    (is _y
     ((lambda (_a _b _y &aux _ab)
        (p _a _b _ab)
        (+ _ab _x _y) )
      1 2 3 ) ) )
```

```
(ft (p _x _y)
    (is _s5 (+ _x _y))
    `(s _x _y _s5) )
(hn (p _x _y)
    (p 1 2 _aux6)
    (is _y (+ _aux6 _x 3)) )
```

3.2 The bastron transformers

Source-to-source transformations performed by `bastron` are characterized by delivering programs that can always still be understood by the normal RELFUN interpreter. In fact, they map into a RELFUN subset which is usually more simply interpreted and is always more simply compiled by the 'vertical' techniques described in later sections. The following subsections are ordered according to their position in the command hierarchy of section 2, where the `flatten` command (subsection 3.2.2) just serves to prepare the `flatter` command (subsection 3.2.3). Most material in subsections 3.2.2, 3.2.3, and 3.2.7 is taken from [Bol90].

3.2.1 Untupling

Untupling (command: untup) replaces both active and passive n-ary tups by corresponding binary cns nestings, where the empty tup becomes the distinguished constant `nil`. This transformation, similar to list parsing in LISP's `read`, prepares PROLOG-like list allocation in the GWAM.

For example, the ternary tup expression in

```
list3(E) :- & tup(E,E,E).
```

becomes as in

```
list3(E) :- & cns(E,cns(E,cns(E,nil))).
```

while the equivalent tup structure (cf. subsection 3.2.4) in

```
list3(E) :- & [E,E,E].   % list3(E) :- & tup[E,E,E].
```

becomes as in

```
list3(E) :- & cns[E,cns[E,cns[E,nil]]].
```

Sample dialog (untupling of passive head and active body tups):

```
rfi-p> az listn([],_) :- & tup().
rfi-p> az listn([L],E) :- & tup(E|listn(L,E)).
rfi-p> untup
rfi-p> listing
listn(nil,_) :- & nil.
listn(cns[L,nil],E) :- & cns(E,listn(L,E)).
```

3.2.2 Flattening

Flattening (command: `flatten`) replaces embedded subexpressions in the premises (both body and foot) by newly generated variables and associates these with each other through preceding `is`-calls.

For example, one can employ `child` as a binary operator defined by

```
child(john,lucy) :- & ann.
child(john,mary) :- & bob.
```

in calls like `child(P,Q)`, evaluating to P and Q's children. An embedding of such an evaluative formula into another evaluative formula makes the main formula nested. Thus, the `cares` body of the footened form (cf. subsection 3.2.7)

```
parental(P) :- cares(P,child(P,Q)) & true.
```

will be flattened to

```
parental(P) :- _1 is child(P,Q), cares(P,_1) & true.
```

Sample dialog (nested foots would also work):

```
rfi-p> az f(k[]) :- g(h()) & j(k[]).
rfi-p> flatten
rfi-p> listing
f(k[]) :- _1 is h(), g(_1) & j(k[]).
```

3.2.3 Flattering

Flattering (command: `flatter`) acts like `flatten` (cf. subsection 3.2.2) but additionally replaces embedded structures (both in the premises and in the head) by newly generated variables and associates these with each other through preceding `is`-calls.

For example, one can also employ `child` as an undefined binary constructor in structures like `child[P,Q]`, just denoting P and Q's children. An embedding of such a denotative formula into an evaluative formula leaves the main formula flat. Thus, the `cares` body of the footened form

```
parental(P) :- cares(P,child[P,Q]) & true.
```

in subsection 3.2.7 cannot be flattened but it can be flattered to

```
parental(P) :- _1 is child[P,Q], cares(P,_1) & true.
```

Sample dialog (equivalent to `flatten` followed by `flatter` up to variable renaming):

```
rfi-p> az f(k[]) :- g(h()) & j(k[]).
rfi-p> flatter
rfi-p> listing
f(_1) :- _1 is k[], _2 is h(), g(_2), _3 is k[] & j(_3).
```

3.2.4 Tuple- and cons-passivating

Tuple- and cons-passivating (command: `passtup`) replaces active, parenthesized `tup` and `cns` calls containing only constants, variables, and structures/lists by passive, bracketed `tup` structures, i.e. lists, and `cns` structures, respectively.

For example, the `tup` and `cns` expressions in

```
list3(E) :- & tup(E,E,E).
cons2(E) :- & cns(E,E).
```

contain variables only, and thus are `tup`- and `cns`-passivated to structures as, respectively, in

```
list3(E) :- & [E,E,E].                % [E,E,E] shortens tup[E,E,E]
cons2(E) :- & cns[E,E].
```

Sample dialog (only after `flatten` becomes second `tup` passive):

```
rfi-p> az listn([],_) :- & tup().              % [] for 0
rfi-p> az listn([L],E) :- & tup(E|listn(L,E)).  % [L] for n+1
rfi-p> passtup
rfi-p> listing
listn([],_) :- & [].
listn([L],E) :- & tup(E|listn(L,E)).
rfi-p> flatten
rfi-p> listing
listn([],_) :- & [].
listn([L],E) :- _1 is listn(L,E) & tup(E|_1).
rfi-p> passtup
rfi-p> listing
listn([],_) :- & [].
listn([L],E) :- _1 is listn(L,E) & [E|_1].
```

3.2.5 Deanonymization

Deanonymization (command: `deanon`) transforms anonymous variables (PROLOG-like syntax: "_"; LISP-like syntax: "id"), domains (dom-terms), exclusions (exc- terms), and types ("$"-prefixed predicates) to named versions. For doing this new variables are generated replacing each "_"/"id"-occurrence and providing the occurrence-binding (bnd-term) variables for dom/exc-terms and "$"-predicates.

For example, the anonymous terms in the P-pattern of

```
t(A1,A2) :- P is [_,dom[a,b],exc[c],$atom],
            [P,P] is [A1,A2].
```

become as in

```
t(A1,A2) :- P is [_1,bnd[_2,dom[a,b]],bnd[_3,exc[c]],bnd[_4,$atom]],
            [P,P] is [A1,A2].
```

The bnd-variables effect that after further compilation, although both the goals `t([true,a,b,c],[true,a,b,c])` and `t([false,b,a,d],[false,b,a,d])` succeed, the goal `t([true,a,b,c],[false,b,a,d])` correctly fails.

Sample dialog (only the first clause's head is affected):

```
rfi-p> az listn([],_) :- & tup().
rfi-p> az listn([L],E) :- & tup(E|listn(L,E)).
rfi-p> deanon
rfi-p> listing
listn([],_1) :- & tup().
listn([L],E) :- & tup(E|listn(L,E)).
```

3.2.6 Normalizing

Normalizing (command: `normalize`) performs several partial-evaluation-like transformations such as the propagation of passive right-hand sides of is-calls [Kra91].

For example, the constant V-binding in

```
 f(V,W) :- V is a & V.
```

leads to

```
f(a,W) :- & a.
```

Sample dialog (only after `flatter` can `normalize` operate):

```
rfi-p> az f(k[]) :- g(h()) & j(k[]).
rfi-p> normalize
rfi-p> listing
f(k[]) :- g(h()) & j(k[]).
rfi-p> flatter
rfi-p> listing
```

```
f(_1) :- _1 is k[], _2 is h(), g(_2), _3 is k[] & j(_3).
rfi-p> normalize
rfi-p> listing
f(_1) :- _1 is k[], _2 is h(), g(_2) & j(_1).
```

3.2.7 Footening

Footening (command: `footen`) trivially transforms Hornish clauses to footed clauses by introducing the explicit foot `true`. (A `footen` argument can also specify a non-`true` foot.)

For example, the (implicitly `true`-) denotative Hornish rule

```
parental(P) :- cares(P,child[P,Q]).
```

becomes normalized to the following (explicitly `true`-) denotative footed rule:[8]

```
parental(P) :- cares(P,child[P,Q]) & true.
```

Sample dialog (nothing changes since the clause is already footed):

```
rfi-p> az f(k[]) :- g(h()) & j(k[]).
rfi-p> footen
rfi-p> listing
f(k[]) :- g(h()) & j(k[]).
```

4 The classifier

The classifier's task is to extract information (e.g. about the kinds of clauses and variables) from the program (database) that enables the code generator (vertical compiler) to produce efficient RFM (WAM) instructions. This information, often implicit in compilers, is here explicitely represented in the declarative intermediate language Classified Clauses; for this the classifier extends normal **RELFUN** source clauses with numerous declarations on different levels of description. The following short introduction is based on the current implementation status of the Classified Clauses. A more detailed introduction of an earlier version is presented (in German) in [Kra90]. This section briefly describes the Classified Clauses by stepwise refinement; in section 4.7 the description grammar is given in an EBNF syntax.

In Classified Clauses we distinguish six levels of description, namely the database, procedure, clause, chunk, literal, and term levels. A database consists

---

[8] If performed indiscriminately, footening prevents the last-call optimization in the WAM (here, `parental` cannot just jump to, or `execute`, `cares` since it still has to `put_constant true`). In order to avoid this, footening should, in practice, only be performed on Hornish rules for which it cannot be assured that the last premise (here, `cares`) on success will itself return `true`. If, however, this '`true`-return' property can be established for a Hornish rule, it should be 'foot-optimized', i.e. transformed into a footed rule reusing the last (relational) premise as its (functional) foot (here obtaining `parental(P) :-& cares(P,child[P,Q]))`. While in general this requires global analysis, for the important special case of tail-recursion optimization the analysis can be confined to individual procedures. Benchmark results for the latter case can be found in [Hei91].

of an unordered set of procedures each consisting of an ordered set of clauses. All clauses of one procedure have the same name and arity. Name and arity yield the procedure name 'name/arity'. For example, the clause `foo(V,W)` belongs to the procedure `foo/2`.

The Classified Clauses for a **RELFUN** program (database) are accordingly defined as follows:

classified_database ::= (**db**[9] {classified_procedure}*)

4.1 Procedure level

Syntax:

classified_procedure ::= (**proc** procedure_name clause_count indexing {clause_classification}+)

Description:

proc Each description of a procedure starts with the tag **proc**.

procedure_name The name and the arity of clauses yield the procedure name.

clause_count Clause_count gives the number of clauses belonging to the procedure.

indexing Indexing information for the procedure.

Example:

Prolog-like source:

```
foo(...).
foo(...) :- . . . .
. . .
```

Lisp-like source:

```
(hn (foo ...))
(ft (foo ...) . . .)
. . .
```

Classified Clauses:

```
(db (proc foo/2 2
          indexing
          clause_classification
          clause_classification)
    . . .)
```

---

[9] The db tag is omitted in the current implementation

Remark:
It is planned for the future to extend the description of a procedure by information about the modes of the arguments in all feasible calls to the procedure. In this way it should be possible that, on the one hand, the user can declare the modes and, on the other hand, a mode interpreter can compute the modes automatically. Thus the mode interpreter could check the consistency of the modes generated by the user in exactly the same way.

4.2 Indexing

Syntax:

indexing ::= (**indexing** [iblock])
iblock ::= pblock | sblock
pblock ::= (**pblock** rblock {sblock | 1block}$^{+}$)
rblock ::= (**rblock** clauses {arg-col}$^{+}$)
clauses ::= (**clauses** {clause-number}$^{+}$)
arg-col ::= (**arg** arg-number {base-type}$^{+}$)
base-type ::= const | struct | var
const ::= (**const** symbol)
struct ::= (**struct** symbol arity)
var ::= (**var** symbol)
1block ::= (**1block** clauses {arg-col}$^{+}$)
sblock ::= (**sblock** rblock seqind [pblock])
seqind ::= (**seqind** {seqind-arg}$^{+}$)
seqind-arg ::= (**arg** arg-number (**info** inhomogenity) constants
structures lists empty-lists [others])
constants ::= (**const** {element}*)
structures ::= (**struct** {element}*)
element ::= (element-name clauses [iblock])
element-name ::= symbol | (symbol arity)
lists ::= (**list** clauses [iblock])
empty-lists ::= (**nil** clauses [iblock])
others ::= (**other** clauses [iblock])

Description:

iblock indexed block

pblock partition block

sblock standard index block

1block block consisting of only one clause

rblock raw block containing the initial data

seqind sequential indexing

arg-col argument column

others (possibly indexed) clauses for elements not occurring in any hash table

Example:

Prolog-like source:

```
foo(alpha,beta).
foo(T,gamma) :- . . . .
. . .
```

Lisp-like source:

```
(hn (foo alpha beta))
(ft (foo _t gamma) . . .)
. . .
```

Classified Clauses:

```
(db (proc foo/2 2
          (indexing
           (sblock
            (rblock
             (clauses 1 2)
             (arg 1 (const alpha) (var t))
             (arg 2 (const beta) (const gamma)) )
            (seqind
             (arg 2
              (info 2)
              (const (beta (clauses 1)) (gamma (clauses 2)))
              (struct) (list) (nil) )
             (arg 1
              (info 1)
              (const (alpha (clauses 1 2)))
              (struct) (list) (nil)
              (other (clauses 2)) ) ) ) )
       . . .)
```

Here we insert a more complete example from a propositional normalizer [Sin93]:

Prolog-like source:

```
norm(X, X) :- literal(X).
norm(or[X, Y], or[X, Y]) :- literal(X), literal(Y).
norm(and[X, Y], and[X, Y]) :- literal(X), literal(Y).
norm(or[X, Y], or[X1, Y]) :- literal(Y), norm(X, X1).
norm(or[X, or[Y, Z]], W) :- norm(or[or[X, Y], Z], W).
norm(or[X, and[Y1, Y2]], or[X1, Y12]) :-
```

```
        norm(X, X1), norm(and[Y1, Y2], Y12).
norm(and[X, Y], and[X1, Y]) :- literal(Y), norm(X, X1).
norm(and[X, and[Y, Z]], W) :- norm(and[and[X, Y], Z], W).
norm(and[X, or[Y1, Y2]], and[X1, Y12]) :- norm(X, X1),
        norm(or[Y1, Y2], Y12).
```

Classified Clauses:

```
(db (proc norm/2 9                  ; norm/2 has 9 clauses
     (indexing
      (sblock
       (rblock                      ; info block for first node
        (clauses 1 2 3 4 5 6 7 8 9) ; of the index tree
        (arg 1         ; possible contents of the first argument
         (var x) (struct or 2) (struct and 2) (struct or 2)
         (struct or 2) (struct or 2) (struct and 2)
         (struct and 2) (struct and 2) )
        (arg 2         ; possible contents of the second argument
         (var x) (struct or 2) (struct and 2) (struct or 2)
         (var w) (struct or 2) (struct and 2)
         (var w) (struct and 2) ) )
       (seqind         ; first node of the index tree
        (arg 1         ; indexing for the first arg
         (info 2)      ; there are 2 possible arguments
         (const)       ; no constant in first arg
         (struct       ; there are heads with struct as 1st arg
                       ; create new node in index tree
          ((or 2)      ; norm(or[..],..)
           (clauses 1 2 4 5 6) ; matches these clauses
           (sblock     ; new node for 2nd-arg indexing
            (rblock    ; information for possible subtree pruning
             (clauses 1 2 4 5 6)
             (arg 2 (var x) (struct or 2)
                    (struct or 2) (var w) (struct or 2)) )
            (seqind
             (arg 2
              (info 1) ; 1 possible arg
              (const)  ; no constant as 2nd arg
              (struct  ; norm(or[..],or[..])
               ((or 2) (clauses 1 2 4 5 6))) ; create try-trust block for
                                             ; these clauses
              (list)   ; no list as 2nd arg
              (nil)    ; no [] as 2nd rg
              (other (clauses 1 5)) ) ) ) ) ; variable as 2nd
          ((and 2)     ; norm(and[..],..)
           (clauses 1 3 7 8 9) ; matches these clauses
           (sblock      ; new node for 2nd-arg indexing
```

```
     (rblock     ; information for possible subtree pruning
      (clauses 1 3 7 8 9)
      (arg 2 (var x) (struct and 2)
             (struct and 2) (var w) (struct and 2)) )
     (seqind
      (arg 2
       (info 1) ; 1 possible arg
       (const)  ; no constant as 2nd arg
       (struct
        ((and 2) (clauses 1 3 7 8 9))) ; create try-trust block for
                                       ; these clauses
       (list)   ; no list as 2nd arg
       (nil)    ; no [] as 2nd arg
       (other (clauses 1 8)) ; variable as 2nd arg
       ))))) ; (struct ...
  (list)        ; no list as 1st arg
  (nil)         ; no list as 1st arg
  (other (clauses 1)) ; variable as 1st arg
  ) ; (arg 1 ...
 (arg 2         ; indexing for the 2nd arg
  (info 2)      ; 2 possible arguments
  (const)       ; no constants
  (struct       ; there are heads with struct as 2nd arg
   ((or 2) (clauses 1 2 4 5 6 8)) ; create try-trust block for
                                  ; norm(..,or[..])
   ((and 2) (clauses 1 3 5 7 8 9))) ; and for norm(..,and[..])
  (list)        ; no list as 2nd arg
  (nil)         ; no [] as 2nd arg
  (other (clauses 1 5 8)) ) ) ) ) ; variable as 2nd arg
. . . )
```

Remark:
For further information about indexing see [Ste93, Sin93, SS92].

4.3 Clause level

Syntax:

clause_classification ::= (clause_type cut_info perm_var_list temp_var_list chunk_sequence)
chunk_sequence ::= head_chunk_fact | head_chunk_rule body_chunk_list
cut_info ::= (**cut-info** cut_type)
perm_var_list ::= (**perm** {global_perm_var_descr}*)
temp_var_list ::= (**temp** {global_temp_var_descr}*)
cut_type ::= **lonely** | **first** | **last** | **general** | **nil**
global_perm_var_descr ::= (variable perm_descr)
global_temp_var_descr ::= (variable temp_descr)
perm_descr ::= (Y-reg_nr use_head (last_chunk last_chunkliteral))
temp_descr ::= (X-reg_nr use_head use_premise)

Description:

clause_type The clause_type describes the kind of clauses, which are distinguished in **rel0**, **fun1den**, **fun1eva**, **fun*den**, **fun*eva**. We give the type **rel0** to a hn-clause without any body literal. Thus **rel0** tags an ordinary fact, as known from PROLOG. The "1" in the types **fun1den** and **fun1eva** indicates that the clause contains only one **chunk**. Hence "*" means the clause contains two or more **chunks**. "den" stands for denotative foot and "eva" for evaluative foot. It should be noted that an hn-clause with an evaluative last body literal still is a "den"-like clause, because hn-clauses implicitly return the value true and not the value of their last premise

cut_info (Information about the occurrence of a cut in the clause) The cut_info contains exactly one argument, cut_type, which maps directly to the corresponding **GWAM**-instructions (see section 7). The cut_type argument is **nil** if there is no cut. Since currently RELFUN clauses always return a value, only **first** and **general** are in use.

perm_var_list (Global information about the permanent variables of the clause) An element of the perm_var_list is a pair of the form: (variable perm_descr). The perm_descr is a 3-tuple describing a) where the variable has to be located in the local environment in order to make optimum environment trimming, b) the occurrences in the head literal (a list of argument positions), and c) the last occurrence (the last **chunk** and the last literal in this **chunk**) of the variable in the clause.

temp_var_list (Global information about the temporary variables in the clause) The temp_var_list describes a) which register (or X-reg_nr) has to be assigned to the temporary variable for register optimization on the machine level, b) the occurrence in the head literal (or use_head), and c) the call literal (or use_premise). A temporary variable occurs only in one **chunk** by definition; in this way the call literal is unique and it is possible that neither use_head nor use_premise are different from the empty list nil.

Example:

Prolog-like source:

```
foo(alpha,beta).
foo(T,gamma) :- bar(T,P) !& bar(P,Q).
. . .
```

Lisp-like source:

```
(hn (foo alpha beta))
(ft (foo _t gamma) (bar _t _p) ! (bar _p _q))
. . .
```

Classified Clauses:

```
(db (proc foo/2 2 (indexing .  .  .)
          (rel0                 ; hn-clause (foo alpha ...)
                                ; without body goals
                (cut-info nil)  ; there is no cut
                (perm)          ; there are no permanent variables
                (temp)          ; there are no temporary variables
                (chunk .  .  .))   ; head_chunk_fact


          (fun*eva            ; the ft-clause (foo _t ...). The
                              ; clause contains two small chunks
                              ; and an evaluative foot calling bar/2
                (cut-info general)
                (perm (_p (1 nil (2 1)))); Permanent variable _p.
                              ; _p is assigned to the Y-reg 1 in the
                              ; local environment. _p doesn't occur
                              ; in the head. Its last occurrence is
                              ; in the second chunk and as the first
                              ; literal in the chunk.

                (temp (_t (1 (1) (1))) ; The temporary variable _t.
                               ; _t is assigned to the X-reg 1. It
                               ; has an arg-1 occurrence in the head.
                               ; Its call literal in the chunk is
                               ; in the argument position 1.
                      (_q (2 nil (2)))) ; _q is assigned to register 2
                               ; because its occurrence in the call
                               ; literal is at argument position 2.
                               ; It has no head occurrence.
                (chunk .  .  .)    ; head_chunk_rule
                (chunk .  .  .))  ; body_chunk
      .  .  . )
```

4.4 Chunk level

Syntax:

head_chunk_fact ::= (**chunk** (head_literal {chunk_guard}*) chunk_descr)
head_chunk_rule ::= (**chunk** (head_literal {chunk_guard}* first_premise_literal) chunk_descr)
body_chunk_list ::= {body_chunk}* [(({chunk_guard}*) chunk_descr)]
body_chunk ::= (**chunk** ({chunk_guard}* call_literal) chunk_descr)
call_literal ::= literal_classification | lispcall_classification
chunk_guard ::= builtin | passive_term
chunk_descr ::= (lu_reg ({(variable permvar_uselit_list)}*))
permvar_uselit_list ::= ({arg_nr}$^{+}$)

Description:

body_chunk A chunk is a 2-argumented structure composed of the tag chunk, a list of denotative literals called chunk_guards with an additional evaluative literal called call_literal as the last element, and some information about the **chunk** called chunk_descr.

head_chunk_fact If there are no call literals in the body of the clause, then the clause contains only one **chunk** ending with a denotative literal. We call this kind of **chunk** head_chunk_fact. In fact, all clauses with type **rel0** or **fun1den** are constructed with only the head_chunk_fact.

head_chunk_rule If there is at least one call literal in the clause, then the first **chunk** ends with a call literal (first_premise_literal). All clauses with types different from **rel0** and **fun1den** have a head_chunk_rule as their first **chunk**.

chunk_descr The classifier computes optimized register assignments for temporary variables. The information lu_reg tells the code generator which register is the last one used by the classifier. For example the code generator has to take register numbers higher than lu_reg for handling the permanent variables in the **chunk**. The pair (variable permvar_uselit_list) tells the code generator where the permanent variables occur in the call_literal of the **chunk**.

Example:

Prolog-like source:

```
foo(alpha,beta).
foo(T,gamma) :- bar(T,P) !& bar(P,Q).
.  .  .
```

Lisp-like source:

```
(hn (foo alpha beta))
(ft (foo _t gamma) (bar _t _p) ! (bar _p _q))
.  .  .
```

Classified Clauses:

```
(db (proc foo/2 2 (indexing ...)
      (rel0                    ; hn-clause without body goals
            (cut-info nil)
            (perm)
            (temp)
            (chunk             ; The tag for the first chunk.
                 (head_literal) ; There exists only the head literal
                 nil) )        ; There is no need for any chunk_descr
```

```
      (fun*eva
           (cut-info general)
           (perm (_p (1 nil (2 1))))
           (temp (_t (1 (1) (1)))
                (_q (2 nil (2))))
           (chunk       ; The tag for the first chunk.
                ((usrlit ...))    ; head_literal  first_premise_lite
                (2 ((_p (2)))) ); lu_reg = 2 because of the arity
                            ; of the first_premise_literal. The
                            ; permanent variable _p occurs at
                            ; position 2 in the call_literal.
           (chunk           ; The tag for the second chunk.
                ((usrlit ...))    ; there is only a call_literal.
                (2 ((_p (1)))))) ; _p occurs at position 1
                                 ; in the call_literal.
  . . .)
```

4.5 Literal level and argument level

Syntax:

literal_classification ::= (**usrlit** (functor arglist_classification) literal_descr)
lispcall_classification ::= (lispcall_type (lisp-builtin arglist_classification)
    lispcall_descr)
builtin ::= **unknown** | is_primitive | (**refl-Xreg** lhs_term)
arglist_classification ::= {term_classification}*
term_classification ::= constant_classification | variable_classification
    | structure_classification
is_primitive ::= (**is** lhs_term rhs_term)
lhs_term ::= constant_classification | variable_classification
rhs_term ::= term_classification
constant_classification ::= constant_name
variable_classification ::= (variable local_var_descr)
structure_classification ::= '(functor arglist_classification)
    | (**inst** (functor arglist_classification))
local_var_descr ::= (occurrence saveness var_class)
literal_descr ::= (arity env_size arg_seq)
lispcall_descr ::= (arity env_size arg_seq)

Description:

term_classification A term is a denotative literal. The inst_op (“‘” or “inst”) indicates that a literal is a denotative (sometimes called passive) one.

local_var_descr A variable is locally described (with respect to all its occurrences in the clauses) by the local_var_descr. It is a list of three elements (occurrence saveness var_class). The occurrence can be first, nonfirst, or reuse. While the meaning of first and nonfirst is intuitively clear, reuse means that the classifier has assigned a register to more than one temporary variable. If a variable occurs first it gets the information reuse (instead of first) when the register was assigned to another temporary variable before in the same **chunk**. This is more an information for the user than for the code generator. Because of the different possible references of a variable, we describe the different reference states by the information saveness. The saveness is distinguished into global (a reference to the heap), safe (a reference to a caller environment or to the heap), and unsafe (a possible reference to the local environment). The information var_class tells the code generator whether the variable is temp or perm.

literal_descr The arity gives the number of arguments in the literal.

env_size denotes how many permanent variables have to survive the call to the literal. The Y-register assignment in the permvar_list has been done in a way that the env_size is as small as possible.

arg_seq is a list that tells the code generator in which order the argument positions have to be represented by **GWAM** instructions. It is possible that some arguments need no instructions. A missing argument position in arg-seq indicates such a case.

Example:

Prolog-like source:

```
foo(alpha,beta).
foo(T,gamma) :- bar(T,P) !& bar(P,Q).
. . .
```

Lisp-like source:

```
(hn (foo alpha beta))
(ft (foo _t gamma) (bar _t _p) ! (bar _p _q))
. . .
```

Classified Clauses:

```
(db (proc foo/2 2 (indexing ...)
      (rel0
            (cut-info nil)
            (perm) (temp)
            (chunk
                   ((usrlit (foo alpha beta)
                            (2 0 (1 2)))) ; The literal foo has 2
                                          ; arguments. The env_size is 0.
                                          ; Use the order given in
                                          ; arg_seq (1st: alpha, 2nd:
```

```
                                  ; beta.
                nil))             ; No chunk description needed
   (fun*eva
         (cut-info general)
         (perm (_p (1 nil (2 1))))
         (temp (_t (1 (1) (1))) (_q (2 nil (2))))
         (chunk
                ((usrlit (foo (_t (first safe temp)); _t occurs
                                  ; first and is safe because
                                  ; it has a reference to the
                              gamma); caller's environment
                         (2 1 (2))); _t needs no instruction
                                   ; since it stays first arg
                 (usrlit (bar (_t (nonfirst safe temp))
                              (_p (first unsafe perm)))
                                  ; _p is potentially unsafe
                         (2 1 (2)))); As above!
                                  ; No instruction for _t
                (2 ((_p (2)))) )
         (chunk ((cutlit (cut) (0 1 nil))) (0 nil))
         (chunk
                ((usrlit (bar (_p (nonfirst unsafe perm))
                              (_q (first unsafe temp)))
                 (2 0 (1 2))))
                (2 ((_p (1)))))))
. . .)
```

Remark:
The WAM-instruction meaning of the Classified Clauses is described in paragraph 5, where an introduction to the code generator is given. The code generator takes as input the Classified Clauses for **RELFUN** and produces the **GWAM** code. Therefore, in paragraph 5 you can find more detailed information on how the added annotations are used for code generation.

4.6 An example with structures

We consider an example showing in which way structures are represented in the Classified Clauses. The first step we show is the flattening and normalizing that precedes (as part of the horizon command, cf. section 3.2) the compilation before classified clauses are generated (see [Kra91] and section 2).

Prolog-like source:

```
bar(R,S).
fie(f[b],f[b],b) :- W is g[f[b]] & bar(b,W).
```

Leads after flattering and normalizing to:

```
bar(R,S).
fie(_3,_3,b) :- _3 is f[b], W is g[_3] & bar(b,W).
```

Lisp-like source:

```
(hn (bar _r _s))
(ft (fie _3 _3 b)
    (is _3 '(f b))
    (is _w '(g _3))
    (bar b _w) )
```

Classified Clauses:

```
(db (proc bar/2  1
   (indexing)                  ; no indexing
   (rel0                       ; bar/2 is an hn-fact
        (cut-info nil)         ; no cut
        (perm)                 ; No permanent variables
        (temp (_r (1 (1) nil)) ; 2 temporary variables
              (_s (2 (2) nil)))
        (chunk
              ((usrlit (bar (_r (first safe temp))
                            (_s (first safe temp)))
                       (2 0 (1 2)) )); Proposed instructions for position 1 and
              nil)))          ; 2, but the code generator will make it better

; Start of the description of the next procedure

 (proc fie/3  1
      (indexing)                  ; no indexing
      (fun1eva                    ; A one-chunk rule with an evaluative foot
           (cut-info nil)
           (perm)
           (temp (_3 (1 (2 1) nil)) ; the variable _3 has no occurrence
                                  ; in the call_literal of its chunk
                 (_w (2 nil (2))))
           (chunk ((usrlit (fie (_3 (first safe temp))
                                (_3 (nonfirst safe temp))
                                b) ; A constant gets no further description
                           (3 0 (3 1 2)) ) ; Generate code for the constant first!
                  (is (_3 (nonfirst global temp))
                      '(f b))     ; A chunk guard gets no further description
                  (is             ; All is-primitives are used denotatively
                      (_w (first unsafe temp)) ; in the Classified Clauses
                      '(g (_3 (nonfirst safe temp))) ) ; The structure g/2
                                                       ; beginning with "'"
                  (usrlit (bar b
                                (_w (nonfirst unsafe temp)))
```

```
                 (2 0 (1)))); No instruction for _w necessary because
                     ; the register 2 is assigned to it
            (3 nil)))))      ; lu_reg = 3, because of the literal foo/3
```

4.7 EBNF syntax for Classified clauses

classified_database ::= (**db** {classified_procedure}*)
classified_procedure ::= (**proc** procedure_name clause_count indexing {clause_classification}$^{+}$)
indexing ::= (**indexing** [iblock])
iblock ::= pblock | sblock
pblock ::= (**pblock** rblock {sblock | 1block}$^{+}$)
rblock ::= (**rblock** clauses {arg-col}$^{+}$)
clauses ::= (**clauses** {clause-number}$^{+}$)
arg-col ::= (**arg** arg-number {base-type}$^{+}$)
base-type ::= const | struct | var
const ::= (**const** symbol)
struct ::= (**struct** symbol arity)
var ::= (**var** symbol)
1block ::= (**1block** clauses {arg-col}$^{+}$)
sblock ::= (**sblock** rblock seqind [pblock])
seqind ::= (**seqind** {seqind-arg}$^{+}$)
seqind-arg ::= (**arg** arg-number (**info** inhomogenity) constants structures lists empty-lists [others])
constants ::= (**const** {element}*)
structures ::= (**struct** {element}*)
element ::= (element-name clauses [iblock])
element-name ::= symbol | (symbol arity)
lists ::= (**list** clauses [iblock])
empty-lists ::= (**nil** clauses [iblock])
others ::= (**other** clauses [iblock])
clause_classification ::= (clause_type cut_info perm_var_list temp_var_list chunk_sequence)
chunk_sequence ::= head_chunk_fact | head_chunk_rule body_chunk_list
cut_info ::= (**cut-info** cut_type)
head_chunk_fact ::= (**chunk** (head_literal {chunk_guard}*) chunk_descr)
head_chunk_rule ::= (**chunk** (head_literal {chunk_guard}* first_premise_literal) chunk_descr)
body_chunk_list ::= {body_chunk}* [(({chunk_guard}*) chunk_descr)]
body_chunk ::= (**chunk** ({chunk_guard}* call_literal) chunk_descr)
chunk_descr ::= (lu_reg ({(variable permvar_uselit_list)}*))
head_literal ::= literal_classification
first_premise_literal ::= call_literal
call_literal ::= literal_classification | lispcall_classification
chunk_guard ::= builtin | passive_term
passive_term ::= term_classification
permvar_uselit_list ::= ({arg_nr}$^{+}$)
literal_classification ::= (**usrlit** (functor arglist_classification) literal_descr)
lispcall_classification ::= (lispcall_type (lisp-builtin arglist_classification) lispcall_descr)
builtin ::= **unknown** | is_primitive | (**refl-Xreg** lhs_term)
arglist_classification ::= {term_classification}*

term_classification ::= constant_classification | variable_classification
| structure_classification
is_primitive ::= (**is** lhs_term rhs_term)
lhs_term ::= constant_classification | variable_classification
rhs_term ::= term_classification
constant_classification ::= constant_name
variable_classification ::= (variable local_var_descr)
structure_classification ::= '(functor arglist_classification)
| (**inst** (functor arglist_classification))
perm_var_list ::= (**perm** {global_perm_var_descr}*)
temp_var_list ::= (**temp** {global_temp_var_descr}*)
literal_descr ::= (arity env_size arg_seq)
lispcall_descr ::= (arity env_size arg_seq)
global_perm_var_descr ::= (variable perm_descr)
global_temp_var_descr ::= (variable temp_descr)
perm_descr ::= (Y-reg_nr use_head (last_chunk last_chunkliteral))
temp_descr ::= (X-reg_nr use_head use_premise)
local_var_descr ::= (occurrence saveness var_class)
clause_type ::= **rel0** | **fun1den** | **fun1eva** | **fun*den** | **fun*eva**
lispcall_type ::= **cl-func** | **cl-pred** | **cl-extra**
Y-reg_nr ::= reg_nr
X-reg_nr ::= reg_nr
last_chunk ::= chunk_nr
last_chunkliteral ::= lit_nr
use_head ::= ({reg_nr}*)
use_premise ::= ({reg_nr}*)
arg_seq ::= ({arg_nr}*)
lu_reg ::= reg_nr
occurrence ::= **first** | **nonfirst** | **reuse**
saveness ::= **global** | **safe** | **unsafe**
var_class ::= **perm** | **temp**
variable ::= _name | (**vari** name)
procedure_name ::= name/arity
functor ::= name
lisp-builtin ::= lisp-fcts | lisp-preds | lisp-extras
lisp-fcts ::= ;;;;; **RELFUN** supported LISP functions
lisp-preds ::= ;;;;; **RELFUN** supported LISP predicates
lisp-extras ::= ;;;;; **RELFUN** supported LISP functions with side effects
constant_name ::= name
clause_count ::= cardinal
arg_nr ::= cardinal
reg_nr ::= cardinal
chunk_nr ::= cardinal
lit_nr ::= cardinal0
env_size ::= cardinal0
arity ::= cardinal0

name ::= letter {letter | digit0}*
cardinal ::= digit {digit0}*
cardinal0 ::= **0** | cardinal
letter ::= **a** | **b** | ... | **z**
digit ::= **1** | **2** | ... | **9**
digit0 ::= **0** | digit

5 The code generator

The basic idea of the code generator is to keep it as simple as possible to allow an easy replacement of the **GWAM** by another abstract machine. The classified clauses should be considered as a 'machine-independent' representation of **RELFUN** procedures. It was not necessary to modify the code generator when proceeding from Nystrøm's WAM to our **GWAM** and C-based emulators.

The internal program structure of the code generator resembles the structure of the EBNF syntax. Therefore, in the following we give the EBNF syntax and the corresponding LISP functions.

The idea is to associate with each nonterminal symbol a function returning code for the corresponding construct; the returned code is appended to the already existing code. This ensures a (more or less) functional structure of the code generator. To avoid possible performance problems of the code generator, all calls to the expensive append are encapsulated in the macros `doappend` and `addcode`, where they could be replaced by cheaper `nconc` calls.

In this section the functions and macros of the code generator will be introduced. The descriptions of the function's parameters will not be given, so the reader should consult the source code, although the variable names should be self-explaining.

The source of the code generator has been written in a very functional style using only a small subset of COMMON LISP, having in mind a simple reimplementation of the code generator in **RELFUN**. Thus, we make extensive use of CONDs instead of using ecase, jump tables, and other specialities COMMON LISP is offering.

5.1 Software interface

The code generator has two access functions from the outside (from the view of software modules). `(code-gen-proc classified_procedure)` is used to generate WAM code from a classified procedure. This is the function we use from the outside to compile a procedure incrementally.

In the future, the compilation of a single clause may become important for dynamic asserts and retracts. The appropriate function to produce WAM code for a single classified clause is `(code-gen-cc clause_classification)`.

If extensions to the code generator are made, one should ensure that this interface does not change.

In the following, functions for code generation are described. Nonterminals are used as input parameters representing the argument type. The right arrows prefix the returned value of the system, which is often represented by nonterminal symbols. The symbols in bold case are the terminal symbols.

5.2 classified_procedure

classified_procedure ::= (**proc** procedure_name clause_count indexing {clause_classification}$^+$)

- (s-cg-proc-id classified_procedure)
  → **proc**
  remark: s-cg = selector for code generator

- (s-cg-procedure_name classified_procedure)
  → procedure_name

- (s-cg-clause_count classified_procedure)
  → clause_count

- (s-cg-clause_classifications classified_procedure)
  → list of clause_classification(s)

- (code-gen-proc classified_procedure)
  → **GWAM** code for the procedure. This procedure is responsible for generating try/retry/trust instructions.

5.3 indexing

indexing ::= (**indexing** [iblock])

. . .

- (icl.s-iblock-from-class-proc classified_procedure)
  → sblock | pblock
  remark: icl = indexing classifier part

- (icl.s-iblock-type iblock)
  → pblock | sblock

- (icl.s-rblock-from-pblock pblock)
  → rblock

- (icl.s-iblock-list-from-pblock pblock)
  → list of sblock | lblock

- (icl.s-rblock-from-sblock sblock)
  → rblock

- (icl.s-seqind-arg-list-from-sblock sblock)
  → list of seqind-arg

- (icl.s-iblock-from-sblock sblock)
  → pblock

- (icl.s-clause-from-lblock lblock)
  → clause-number

- (icl.s-arg-col-list-from-lblock lblock)
  → list of arg-col

- (icl.s-clauses-from-rblock rblock)
  → list of clause-number

- (icl.s-arg-col-list-from-rblock rblock)
  → list of arg-col
- (icl.s-arg-no-from-arg-col arg-col)
  → arg-number
- (icl.s-it-list-from-arg-col arg-col)
  → list of base-type
- (icl.s-arg-no-from-seqind-arg seqind-arg)
  → arg-number
- (icl.s-info-from-seqind-arg seqind-arg)
  → (**info** inhomogenity)
- (icl.s-constant-list-from-seqind-arg seqind-arg)
  → constants
- (icl.s-structure-list-from-seqind-arg seqind-arg)
  → list of elements of structures
- (icl.s-list-from-seqind-arg seqind-arg)
  → lists
- (icl.s-nil-from-seqind-arg seqind-arg)
  → empty-lists
- (icl.s-other-from-seqind-arg seqind-arg)
  → others
- (icl.s-var-from-raw-seqind-arg seqind-arg)
  → lists
- (iif.mk-tree clause_classification)
  → produces indexing trees for further use by the code generator
  remark: iif = indexing interface

5.4 clause_classification

clause_classification ::= (clause_type cut-info perm_var_list temp_var_list chunk_sequence)
chunk_sequence ::= head_chunk_fact | head_chunk_rule body_chunk_list

- (s-cg-clause_typ clause_classification)
  → clause_type
- (s-cg-cut_info clause_classification)
  → cut-info
- (s-cg-perm_var_list clause_classification)
  → perm_var_list

- (s-cg-temp_var_list clause_classification)
  → temp_var_list

- (s-cg-chunks clause_classification)
  → list of head_chunk_fact or list of head_chunk_fact or list of head_chunk_rule
  body_chunk_rule.

- (code-gen-cc clause_classification)
  → **GWAM** code for a classified clause. This function has to cope with **rel0**, **fun1den**, **fun1eva**, **fun*den** and **fun*eva** and with setting up an appropriate environment.

5.5 head_chunk_fact, head_chunk_rule, body_chunk

head_chunk_fact ::= (**chunk** (head_literal {chunk_guard}*) chunk_descr)
head_chunk_rule ::= (**chunk** (head_literal {chunk_guard}* first_premise_literal) chunk_descr)
body_chunk_list ::= {body_chunk}* [(({chunk_guard}*) chunk_descr)]
body_chunk ::= (**chunk** ({chunk_guard}* call_literal) chunk_descr)

Let chnk be an abbreviation for *head_chunk_fact*, *head_chunk_rule* or *body_chunk*.

- (s-cg-chunk_id chnk)
  → **chunk**

- (s-cg-chunk_descr chnk)
  → chunk_descr

- (s-cg-chunk_head_literal chnk)
  → head_literal

- (s-cg-chunk_hd_cgfpl head_chunk_rule)
  → list: ((chunk_guard/s) first_premise_literal)
  remark: cgfpl = chunk guard, first premise literal

- (s-cg-chunk-bd_cgcl body_chunk)
  → ((chunks_guard/s) call_literal)
  remark: cgcl = chunk guard, call literal

- (code-gen-hdchunk perms temps chunk callexeflg deallocflg chunknr)
  This function returns code for the first chunk in the clause. One may notice that this function is very similar to code-gen-chunk below, although further enhancements (indexing, global compilation) may result in a complete reformulation of that function, whereas code-gen-chunk is likely to keep the same.

- (code-gen-chunk perms temps chunk callexeflg deallocflg chunknr)
  Returns WAM code for a chunk to be found in the body.

5.6 chunk_descr

chunk_descr ::= (lu_reg ({(variable permvar_uselit_list)}*))

- s-cg-chunk_lu_reg (chk_descr)
  → lu_reg
- s-cg-chunk_vpul (chk_descr)
  → list of (variable permvar_uselit_list)

5.7 literal_classification

literal_classification ::= (**usrlit** (functor arglist_classification) literal_descr)

- (s-cg-usrlit_id literal_classification)
  → **usrlit**
- (s-cg-literal_descr literal_classification)
  → literal_descr
- (s-cg-fac_list literal_classification)
  → (functor arglist_classification)
  remark: fac = **f**unctor **a**rglist **c**lassification
- (s-cg-functor fac)
  → functor
- (s-cg-arglist_classification fac)
  → arglist_classification
- (code-gen-head perms temps fac arg_seq)
  Generates code for the first literal in the clause.
  - (code-gen-head-arg place temps arg)
    Generates code for an argument place in the first literal in the clause.
  - (code-gen-head-temp place temps arg)
    Generates code for an X-variable in the first literal of a clause.
  - (code-gen-head-perm place temps arg)
    Generates code for a Y-variable in the first literal of a clause.
- (code-gen-tail perms temps arity permcnt fac callexeflg deallocflg cnknr litnr arg_seq)
  Generates code for the literals except the first in the clause.
  - (code-gen-tail-arg place perms temps arg chknr litnr)
    Generates code for an argument place in the literals except the first in the clause.
  - (code-gen-tail-temp place temps arg)
    Generates code for an X-variable in the body literals of a clause.
  - (code-gen-tail-perm place perms arg chknr litnr)
    Generates code for the literals except the first in the clause.

5.8 variable_classification, local_var_descr

variable_classification ::= (variable local_var_descr)
local_var_descr ::= (occurrence saveness var_class)

- (s-cg-local-var-descr variable_classification)
  → local_var_descr
- (s-cg-local_var_occurrence variable_classification)
  → local_var_occurrence
- (s-cg-local_var_saveness variable_classification)
  → local_var_saveness
- (s-cg-local_var_class variable_classification)
  → local_var_class

5.9 Global variables

- Emulator-related variables
  - *user-variables*
    Contains the user's variables when a query is issued.
  - *registers*
    The `define-register` function adds each register to this list, causing the debugger to output the variables of this list.
  - *read-mode*
    This is a global flag in the machine indicating the read/write status, which is used in the unify instructions.
  - *emu-debug*
    This flag determines whether the emulator is in a debugging state or will just run through the code. It can have the following values:
    - `:interactive` the emulator performs single steps
    - T the emulator shows all executed instructions without interaction
    - **nil** if no debugging is demanded
- code generator-related variables
  - *lureg*
    This variable determines which X-registers can be used by the code generator without any interference with the classifier's allocations.
  - y-x-usage-list
    An assoc-list mapping Y variables to X-registers.

5.10 perm_var_list, temp_var_list

perm_var_list ::= (**perm** {global_perm_var_descr}*)
temp_var_list ::= (**temp** {global_temp_var_descr}*)
global_perm_var_descr ::= (variable perm_descr)
global_temp_var_descr ::= (variable temp_descr)

- (s-cg-perm_var global_perm_var_descr)
  → variable
- (s-cg-perm_descr global_perm_var_descr)
  → perm_descr
- (s-cg-temp_var global_temp_var_descr)
  → variable
- (s-cg-temp_descr global_temp_var_descr)
  → temp_descr

5.11 perm_descr, temp_descr

perm_descr ::= (Y-reg_nr use_head (last_chunk last_chunkliteral))
temp_descr ::= (X-reg_nr use_head use_premise)

- (s-cg-perm_y_nr perm_descr)
  → Y-reg_nr
- (s-cg-perm_use_head perm_descr)
  → use_head
- (s-cg-perm_last_literal perm_descr)
  → last_chunkliteral
- (s-cg-temp_x_nr temp_descr)
  → X-reg_nr
- (s-cg-temp_use_head temp_descr)
  → use_head
- (s-cg-temp_use_premise temp_descr)
  → use_premise

5.12 literal_descr

literal_descr ::= (arity env_size arg_seq)

- (s-cg-arity literal_descr)
  → arity
- (s-cg-env_size literal_descr)
  → env_size
- (s-cg-arg_seq literal_descr)
  → arg_seq

5.13 lispcall_type, lispcall_classification

lispcall_classification ::= (lispcall_type (lisp-builtin arglist_classification) lispcall_descr)
lispcall_type ::= **cl-func** | **cl-pred** | **cl-extra** | **cl-relf**

- (cg-lispcall-p lispcall_classification)
  → t, if it is an external LISP call, nil otherwise
- (cg-lispcall-fun lispcall_classification)
  → lisp-function
- (cg-lispcall-args lispcall_classification)
  → arglist_classification

5.14 arglist_classification, term_classification, constant_classification

arglist_classification ::= {term_classification}*
term_classification ::= constant_classification | variable_classification
| structure_classification
constant_classification ::= constant_name
variable_classification ::= see 5.8
structure_classification ::= '(functor arglist_classification)
| (**inst** (functor arglist_classification))

- (cg-inst-p term_classification)
  → t, if argument is an instantiation operator, nil otherwise
- (cg-s-inst-functor term_classification) (already knowing term is inst-op)
  → functor
- (cg-s-inst-funargs term_classification) (already knowing term is inst-op)
  → arglist_classification
- (arg-var-p term_classification)
  → t, if argument is a variable_classification, nil otherwise
- (arg-nil-p arglist_classification)
  → t, if argument is an empty list, nil otherwise
- (arg-const-p arglist_classification)
  → t, if argument is a constant, nil otherwise

5.15 Getting global information on variables

When it is known that a variable with a local description occurs, it is useful to look up the global information. At this level of processing, it is assumed that the code generator already has stored the global X- and Y-variable information in a local variable further referred to as **perms** and **temps**.

- (get_perm_descr arg_var perms)
  get the global information of the permanent variable arg_var.

- (get_temp_descr arg_var perms)
  get the global information of the temporary variable arg_var.

5.16 Obtaining the procedure arity

When coping with a classified_procedure, the arity is needed. This is coded in the procedure_name following the **proc** identifier. However, the arity is coded in an atom symbol unsuitable for (numeric) processing. It is straightforward to extract the number via the COMMON LISP symbol processing functions. The alternative employed here is to use some selectors to get the information from a 'lower' level.

- (s-cg-arity-of-proc proc)
  → arity of the procedure

5.17 The builtins, is_primitive

- (code-gen-is arg1 arg2 perms temps chknr litnr vpul putinl)
  → WAM code for an is-primitive.
- (cg-lispcall-p fac) → t, if fac is a LISP external call.
- (code-gen-cl actual perms temps arity permcnt fac callexeflg deallocflg cnknr litnr arq_seq)
  → WAM code for a LISP external call.
- (code-gen-refl-xreg perms temps arg chknr litnr)
  → WAM code for a refl-xreg builtin. It is used if a value in X1 must be unified with a variable.
    - (code-gen-refl-xreg-perm perms arg chknr litnr)
      → WAM code for a Y-variable in a refl-xreg builtin.
    - (code-gen-refl-xreg-temp temps arg)
      → WAM code for an X-variable in a refl-xreg builtin.

5.18 Y-variable scoreboarding

The idea of Y-variable scoreboarding is to safe memory bandwidth by remembering which Y-variable was already loaded into an X-register. Every time a Y-variable is 'touched', the corresponding X-register is saved as a pair (Y-variable X-register) on an assoc-list named `y-x-usage-list`, which is a global variable meaning that the Y-variable can also be found in an X-register.

The following functions are dealing with Y-variable scoreboarding:

- **(is-y-in-x y-vari y-x-usage-list)**
  This function associates the Y-variable with its X-argument position. If the Y-variable is not in an X-register, the result is `nil`.
- **(add-y-x-list y-vari x-reg y-x-usage-list)**
  This function adds a (Y-variable X-register) pair to the scoreboard.

- **(d_yreg_assoc yreg y-x-usage-list)**
  This is used to eliminate a pair specified by its Y-variable.
- **(d_xreg_assoc xreg y-x-usage-list)**
  This is used to eliminate a pair specified by its X-variable.

6 The GAMA[10]

GAMA, the General Abstract Machine Assembler, is a programming environment supporting the development and integration of abstract machines. In [Sin95], it was used to integrate an existing implementation of the WAM (our development of the NyWAM [Nys], [Hei89]) with the LLAMA [Sin95]).

In the following subsections, the constituents of the **GAMA**,

- the memory organization,
- hash tables, jump tables, and the module system,
- the definition of assembler instructions, and
- the assembler and loader

are described.

6.1 Memory organization

In the **GAMA**, only *one* memory area for all abstract machines exists: the general purpose memory Memory. This memory is managed via a *free list* which contains all areas in Memory which are currently unused. Memory can be allocated and deallocated with the following functions[11]:

- `(gmem.alloc` *n*`)` returns the address of the newly allocated memory area of size *n*
- `(gmem.dealloc` *addr n*`)` deallocates the memory area starting at *addr* with size *n*
- `(gmem.defractionize)` cleans up the free list, i.e. adjacent freed memory areas are collected (after calls to `gmem.dealloc`)

Memory cells can be accessed with the following functions:

- `(gmem.put` *addr x*`)` stores *x* in the cell with address *addr*
- `(gmem.get` *addr*`)` returns the contents of the cell with address *addr*

6.2 Hash tables, jump tables, and the module system

In the **GAMA**, hash tables are simply areas in Memory occupying *three* memory cells for each hash table entry. The use of three cells was motivated by the intended usage of hash tables as *jump tables*: the first cell contains the key (the name of a procedure), the second contains an address (the entry point of the procedure), and the third cell contains further information (concerning the procedure).

The following functions are defined on hash tables:

---

[10] This chapter is completely adopted from chapter 7, "Integrating Abstract Machines: The **GAMA**" in [Sin95].

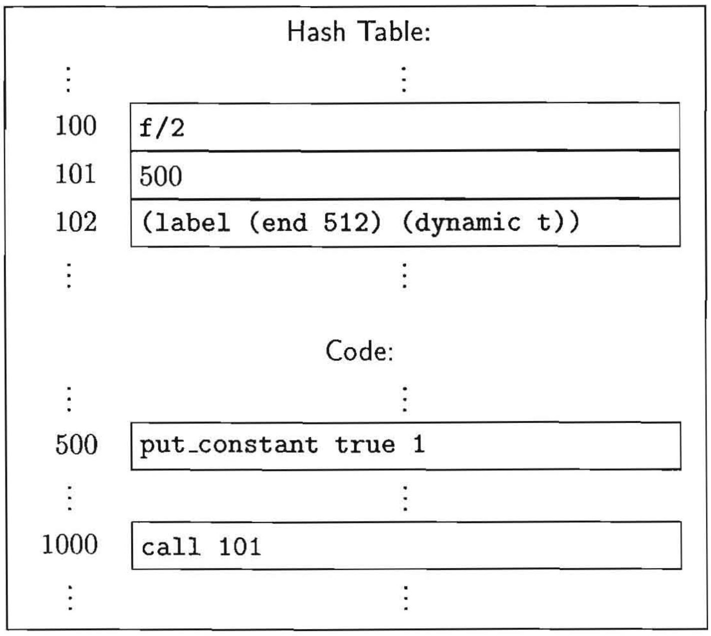
[11] The **GAMA** is implemented in COMMON LISP; in order to avoid name conflicts, function names are preceded by a prefix '*mod.*' indicating that a function belongs to module *mod*, here gmem (we did not use the COMMON LISP package system).

- (`gmht.make-ht` *n*) returns a new hash table handle with *n* entries
- (`gmht.remove-ht` *ht*) removes the hash table *ht*
- (`gmht.put` *ht key a b*) creates a new entry in *ht* for *key*, storing *a* and *b* in it
- (`gmht.get` *ht key*) returns the address (in Memory) of a hash table entry (the *first* address is returned, i.e. the address of the memory cell containing the key)

These hash tables are the basis of the **GAMA** module system: a hash table can be viewed as a name space containing all addresses and further information concerning all procedures of a module.

The reason why addresses are stored independently of the other information is that the hash tables are used as jump tables: a machine instruction like `call` does not have the *name* of a procedure as argument but only the *address* of the second memory cell in the corresponding hash table entry, thus avoiding to look up the address in the hash table at run time.

The following diagram shows how a hash table entry for a procedure `f/2` is used: at the address 1000, a call to `f/2` is expressed as `call 101` where `101` is the address of the memory cell in the hash table which contains the entry point for `f/2`:



Since abstract machines for PROLOG- and LISP-like languages are highly dynamic in that they allow procedures to change even at run time, procedures are not jumped at directly but via jump tables. This has the effect that, if a procedure is changed (recompiled), none of the procedures calling this procedure have to be changed.

6.3 Defining assembler instructions

In the **GAMA**, new assembler instructions for an arbitrary abstract machine are defined with `definstr`. `definstr` expects a COMMON LISP argument list, a type specification for these arguments[12], and the COMMON LISP code defining the instruction.

The following example shows the definition of the **GWAM** instruction `put_constant`:

```
(definstr put_constant (C Ai) (CONST NAT) :standard
  (gwam.put_constant
    (set-argument-reg Ai (constant C))))
```

`gwam.put_constant` is the name of the COMMON LISP function corresponding to the `put_constant` instruction. The keyword `:standard` declares `put_constant` to be a simple instruction. The next example shows a non-standard instruction for which more than one COMMON LISP definition is needed:

```
(definstr call (proc k) (LABEL NAT)
  :static (gwam.call/st
    (set-reg CP (reg P))
    (set-reg CUTP (reg B))
    (if (ref-lessp (reg B) (reg E))
     (set-reg A (ref-plus (reg E) (offset Y) k)))
    (set-reg P proc))
  :dynamic (gwam.call/dy
    (set-reg CP (reg P))
    (set-reg CUTP (reg B))
    (if (ref-lessp (reg B) (reg E))
     (set-reg A (ref-plus (reg E) (offset Y) k)))
    (set-reg P (gmem.get proc))))
```

All instructions expecting a label can be used in two different ways: *statically* and *dynamically*. In the dynamic version, the address corresponding to the label is an entry in a jump table: an additional `gmem.get` is needed to dereference it. The static version does not use a jump table entry but directly uses the real address: dereferencing is not needed. It is used for procedures which will not be changed (like those in the prelude).

6.4 The assembler and loader

In the **GAMA**, assembler and loader are interleaved: in contrast to most assemblers for native machines which first produce a relocatable object file which

---

[12] The available types are: `NAT` for natural numbers, `CONST` for constants, `FUNCTOR` for WAM functor specifications of the form (*name arity*), `FUNCTION` for COMMON LISP functions (e.g. used for builtins), `LABEL` for labels, `VARIABLE` for global variables, `HASHTABLE` for hash tables (used in the WAM switch instructions), and `X` for arbitrary arguments. Additional types can be defined with `gasm.deftype`.

is linked together with other object files by a linker and then loaded into memory for execution, the **GAMA** assembler and loader directly transform assembler code into executable machine code in memory.

In addition to the instructions defined via `definstr`, the **GAMA** assembler handles the following pseudo instructions:

- `.proc` marks the beginning of a procedure; it is mainly used to restrict the scope of local labels thus allowing different procedures to use the same local labels
- `.end` marks the end of a procedure; in addition to restricting the scope of local labels together with `.proc`, it adds the end address of a procedure to the information in the corresponding hash table entry (third cell) in order to allow the procedure to be removed from memory
- `.dynamic` declares the following global labels (the entry points for procedures) to be dynamic (see section 6.3)
- `.static` declares the following global labels to be static
- any *symbol* is taken as a global label
- any *number* or *string* is taken as a local label
- `(.module` *mod*`)` declares all following global labels to be in module *mod*; if this module does not yet exist, it is created
- `(.import-from` *mod* $label_1$ ... $label_n$`)` imports $label_1$ ... $label_n$ from module *mod* (qualified import)
- `(.import-module` *mod*`)` imports all labels from module *mod* (unqualified import)

The following example shows the usage of some of these pseudo instructions and how the assembler and loader transform assembler code into executable machine code in memory.

Example:

The assembler and machine code (with the corresponding hash table entry) for the function

```
fac(0) :- & 1.
fac(N) :- >(N,0) & *(N,fac(1-(N))).
```

is as follows:

Assembler code	Hash table entry and machine code
`.module user`	Hash Table (*for module* `user`):
`.proc`	`21534: fac/1`
`.dynamic`	`21535: 263881`
`fac/1`	`21536: (label`
	`(destroyable t)`
	`(end 263900)`
	`(source ...)`
	`(dynamic t))`
	Code:
`set_index_number 1`	`263881: set_index_number 1`
`switch_on_term`	`263882: switch_on_term`
`"label8965" 2 2`	`263883 263889 263889`
`2 "label8963"`	`263889 263884`
`switch_on_constant 1`	`263883: switch_on_constant 1`
`((0 "label8963")) 2`	`((0 263884)) 263889`
`"label8963"`	
`try 1 1`	`263884: try 263886 1`
`trust 2 1`	`263885: trust 263889 1`
`1`	
`get_constant 0 1`	`263886: get_constant 0 1`
`put_constant 1 1`	`263887: put_constant 1 1`
`proceed`	`263888: proceed`
`2`	
`allocate 1`	`263889: allocate 1`
`get_y_variable 1 1`	`263890: get_y_variable 1 1`
`put_constant 0 2`	`263891: put_constant 0 2`
`cl-pred > 2`	`263892: cl-pred > 2`
`put_y_value 1 1`	`263893: put_y_value 1 1`
`cl-func 1- 1`	`263894: cl-func 1- 1`
`call fac/1 1`	`263895: call 21535 1`
`get_x_variable 2 1`	`263896: get_x_variable 2 1`
`put_y_value 1 1`	`263897: put_y_value 1 1`
`deallocate`	`263898: deallocate`
`cl-func * 2`	`263899: cl-func * 2`
`proceed`	`263900: proceed`
`.end`	

7 The GWAM

The **GWAM** is derived from a LISP-based emulator that was originally obtained from Sven-Olof Nystrøm [Nys], Uppsala University; it was modified to work within our relational-functional compilation approach RFM. This LISP-based implementation has been complemented by two WAM emulators in C: Klaus Elsbernd's rudimentary C emulator [Els90] has now been replaced by Markus Perling's complete first-order emulator. Leaving the layered compiler system in LISP (for flexibility and short turnaround times), but having the emulator in C, seems to be a good combination under UNIX. Thus the **GWAM** is an ideal prototype implementation choice.

7.1 Terminology

'Global Stack' and 'heap' as well as 'local stack', 'stack' and 'runtime stack' are synonyms, an environment and a choice point are portions of the local stack, the push-down list (PDL) is a stack used temporarily by the unification procedure, but it is not needed within the **GWAM**, since this is done recursively in LISP. In most publications the A-registers are assumed to be the same as the X-registers and for those authors assuming disjoint A and X sets of registers the A-regs can be mapped to a single X-register set. Therefore argument registers will be referred herein as X-registers.

7.2 The data structures

The WAM model assumes a tagged memory model. This means that memory locations are 'typed', i.e. that it is possible to tell which datatype is in the memory location. Since registers have neither tags nor addresses, with these it is only possible to handle references (or at most constants) but it is impossible to represent free variables, structures or lists directly. The tagged memory is handled by the following tags:

Tag	Value
empty	undefined
ref	a memory address
struct	a memory address
list	a memory address
const	constant symbol
fun	a list (function-name arity)
trail	a list of references to bound variables

The memory layout is shown in table 1. At the top are the low addresses, increasing downwards.

7.2.1 The local stack

The local stack contains environment and choicepoint frames. An environment must be created in a clause (using the `allocate` instruction) as soon as local variables become necessary.

A choice point is needed if there is more than one clause in a procedure. If a recent goal failed, the next clause must be explored with all argument registers appropriately (re-)set and the variables bound later than the invocation of the current clause restored to an unbound state.

heap (address 0)	← start-of-heap
...	
heap (address n)	← HB
heap (address n+m)	← H
...	
...	
maximum heap address	← start-of-stack−1
local stack	← start-of-stack
...	
environment and choicepoint frames	
...	
local stack	← A
...	
...	
	← memory-size

Table 1: The memory layout of the local and global stacks

previous environment pointer (CE)	← new E
previous continuation pointer (CP)	
cut pointer (CUTP)	
Y-variable$_1$ ... Y-variable$_n$	
	← new A

Table 2: The memory layout of an environment

7.2.2 The heap

The heap holds compound terms. These compound terms may be lists or structures. The H-register points to the top of the heap, whereas the register HB is the (redundant) heap backtrack register used for speeding up references to the old heap pointer.

7.2.3 The trail

Contrary to other implementations the trail is realized as a LISP list. This is possible since no random access may happen on that structure. Either a reference is pushed on the trail (when a binding occurs) or the information is popped sequentially (when backtracking to a certain point occurs).

7.3 The registers

A register defined by `define-register` can be set using `(set-reg` **`register value`**`)` and referenced using `(reg` **`register`**`)`. Currently, there are 1000 X-

X-register$_1$ ... X-register$_n$	
previous environment pointer (BCE)	
previous continuation pointer (BCP)	
previous choice point (B1)	
next clause pointer (BP)	
trail pointer (TR1)	
heap pointer (H1)	← new B
	← new A

Table 3: The memory layout of a choicepoint (backtrack point)

Register	Description	points to	Definition
P	program counter	program code	define-register
CP	continuation pointer	program code	define-register
E	last environment	local stack	define-register
B	last choicepoint	local stack	define-register
A	top of stack	local stack	define-register
TR	trail list		define-register
H	top of heap	heap	define-register
HB	heap backtrack point	heap	define-register
S	structure pointer	heap	define-register
IX	index register		define-register
CUTP	cut pointer	local stack	define-register
X_i	registers	heap, stack	array

Table 4: The registers of the GWAM

registers defined in the array.

7.4 The instructions

The instructions are written in a LISP-like manner. The indexes of X and Y variables start with the index 1. Structures are coded by a list `(fun arity)`. The list structures are coded as nestings of the structure `(cns car cdr)` on the classified clauses representation level. The code generator takes care of these structures, generating the more optimal list instructions.

7.4.1 PUT-instructions

- (put_y_variable Y_{from} X_{to})
- (put_x_variable X_{from} X_{to})
- (put_y_value Y_{from} X_{to})
- (put_x_value X_{from} X_{to})

- (put_unsafe_value Y_{from} X_{to})
- (put_constant C X_{to})
- (put_nil X_{to})
- (put_structure F X_{to})
- (put_list X_{to})

7.4.2 GET-instructions

- (get_x_variable Xn Ai)
- (get_y_variable Yn Ai)
- (get_x_value Xn Ai)
- (get_y_value Yn Ai)
- (get_nil Xi)
- (get_constant C Xi)
- (get_structure F Xi)
- (get_list Xi)

7.4.3 UNIFY-instructions

- (unify_x_variable X_i)
- (unify_y_variable Y_i)
- (unify_void n)
- (unify_x_value X_i)
- (unify_y_value Y_i)
- (unify_x_local_value X_i)
- (unify_y_local_value Y_i)
- (unify_nil)
- (unify_constant C)

7.4.4 Indexing instructions

- (switch_on_term Lconst Lstruct Llist Lnil Lvar)
- (switch_on_constant Len Table Default)
- (switch_on_structure Len Table Default)
- (set_index_number No)

7.4.5 Procedural instructions

- (try L n)
- (retry L n)
- (trust L n)
- (try_me_else L n)
- (retry_me_else L n)
- (trust_me_else_fail n)
- (allocate n)
- (deallocate)
- (proceed)
- (execute proc/n)
- (call proc/n envsize)

7.4.6 Special instructions

- (has-succeeded)
- (has-failed)

7.4.7 Special builtins - cuts and metacall

- (save_cut_pointer)
  This instruction must be generated if there is a cut occurring in the clause except in the first chunk. This implies that there is more than one chunk and an environment must be existent.

- (first_cut)
  This instruction is used when the cut is in the first chunk and the first chunk is no pseudochunk. It contains a call to another procedure and thus is not the only subgoal in the clause.

- (lonely_cut)
  This instruction stands for a clause with a cut at the end of the first and only chunk. (So a call to another procedure is not present.)

- (last_cut)
  last_cut is to be used in a clause, which has a chunk (and hence a call to a procedure) and a cut at the very end of the last (pseudo)-chunk.

- (cut n)
  This instruction represents a cut occurring in a chunk except the first and the last chunk. The parameter n indicates the size of the environment used (for trimming).

7.4.8 LISP interface

Only ground arguments (not variables) can be converted to LISP. The LISP functions are not allowed to return structures (nor variables). All **GAMA**-LISP interface instructions convert *arity* argument registers into a LISP list and apply the function *fun* to this list. Only **RELFUN** tups - but not structures - can be converted.

- (cl-func fun arity)
  This function returns the value obtained from LISP to the argument register X1.
- (cl-pred fun arity)
  This instruction generates a failure if the returned value is `nil`.[13]
- (cl-extra fun arity)
  This instruction is used for side-effect LISP calls.[14]

7.5 User interface of the GWAM

The user may define a procedure using the `definstr` macro. Queries are dynamically compiled by flattening, classifying and generating code for a procedure named '`main/arity`'. The arity of this procedure is determined by the number of variables originally found in the user query.

7.5.1 The debugger control commands

The debugging behavior of the **GWAM** can be controlled by the variable `*emu-debug*`, which is normally set to `nil` to just run through the WAM code. If the user wishes to have WAM debugging information, this global variable may be set to `t` or `:interactive` by the RFE-command `spy`.

If `*emu-debug*` is set to `:interactive`, the following interactions commands may be used:

```
All control commands consist of one character.

E,e             Terminate and go to LISP.
F,f             Generate a fail. (Sometimes this command may
                cause trouble.)
?               Output this Help-Menu.
X,x             Execute until program succeeds.
S,s,newline     Single step execution.
V,v             Output values before single step.
```

---

[13] In the interpreter a `false` is produced, which generates a failure if used as a body premise.

[14] X1 will not be changed.

7.5.2 The debugger display commands

This mode will be enabled by typing `v` in the control mode.

```
All display commands consist of one character.

?               Output this Help-Menu.
X,x             Output n (to be read) argumentregisters X(1)..X(n).
H,h             Output Heap.
R,r             Output all registers except argument registers.
S,s             Output stack.
```

8 A sample session

We consult and compile the well-known naive reverse benchmark, run an `nrev`-query and then demonstrate the usage of the debugger using a simple append-query. Except from the explicit `true` values for successful queries, this does not differ from PROLOG's semantics permitting an easy comparison. Once the debugging principles are thus understood, the reader can also debug functional programs.

```
rfi-p> emul
Collecting modules for the emulator:
sortbase workspace
rfe-p> consult "exa/bench"
Reading file "/home/perling/RELFUN/RFM/demo/exa/bench.rfp"

rfe-p> listing
app([],L,L).
app([H|L1],L2,[H|L3]) :- app(L1,L2,L3).
nrev([],[]).
nrev([H|L1],L3) :- nrev(L1,L2), app(L2,[H],L3).

rfe-p> style lisp
rfe-l> listing
(hn (app (tup) _l _l))
(hn (app (tup _h | _l1) _l2 (tup _h | _l3))
    (app _l1 _l2 _l3) )
(hn (nrev (tup) (tup)))
(hn (nrev (tup _h | _l1) _l3)
    (nrev _l1 _l2)
    (app _l2 '(tup _h) _l3) )
```

The database has been consulted and listed. In the following we do some horizontal transformations and list the result.

```
rfe-l> style prolog
rfe-p> horizon
rfe-p> listing
app(nil,L,L).
app(_1,L2,_2) :- _2 is cns[H,L3], _1 is cns[H,L1], app(L1,L2,L3) & true.
nrev(nil,nil).
nrev(_1,L3) :-
         _1 is cns[H,L1],
         nrev(L1,L2),
         _2 is cns[H,nil],
         app(L2,_2,L3) &
         true.

rfe-p> style lisp
```

```
rfe-l> listing
(hn (app nil _l _l))
(ft (app _1 _l2 _2)
    (is _2 '(cns _h _l3))
    (is _1 '(cns _h _l1))
    (app _l1 _l2 _l3)
    true )
(hn (nrev nil nil))
(ft (nrev _1 _l3)
    (is _1 '(cns _h _l1))
    (nrev _l1 _l2)
    (is _2 '(cns _h nil))
    (app _l2 _2 _l3)
    true )
```

The horizontal transformations are followed by the vertical transformations into WAM code. The resulting code is shown by the `listcode` command. If you want to see the classified clauses, type `listclass`.

```
rfe-l> style prolog
rfe-p> verti
rfe-p> listcode app/3
((set_index_number 1)
 (switch_on_term nil nil 2 1 "label38")
 "label38"
 (set_index_number 3)
 (switch_on_term 1 1 "label39" 1 "label39")
 "label39"
 (try 1 3)
 (trust 2 3)
 1
 (get_nil 1)
 (get_x_value 2 3)
 (put_constant true 1)
 (proceed)
 2
 (allocate 0)
 (get_list 3)
 (unify_x_variable 4)
 (unify_x_variable 5)
 (get_list 1)
 (unify_x_value 4)
 (unify_x_variable 6)
 (put_x_value 6 1)
 (put_x_value 5 3)
 (call app/3 0)
 (put_constant true 1)
```

```
 (deallocate)
 (proceed) )

rfe-p> listcode nrev/2
((set_index_number 1)
 (switch_on_term nil nil 2 1 "label28")
 "label28"
 (set_index_number 2)
 (switch_on_term 2 2 2 "label29" "label29")
 "label29"
 (try 1 2)
 (trust 2 2)
 1
 (get_nil 1)
 (get_nil 2)
 (put_constant true 1)
 (proceed)
 2
 (allocate 3)
 (get_y_variable 3 2)
 (get_list 1)
 (unify_y_variable 2)
 (unify_x_variable 3)
 (put_y_variable 1 2)
 (put_x_value 3 1)
 (call nrev/2 3)
 (put_list 2)
 (unify_y_value 2)
 (unify_nil)
 (put_unsafe_value 1 1)
 (put_y_value 3 3)
 (call app/3 0)
 (put_constant true 1)
 (deallocate)
 (proceed) )
```

We are now finished compiling the database. Next we perform an nrev-query.

```
rfe-p> nrev([1,2,3],X)
true
X=[3,2,1]
rfe-p> more
unknown
```

Now we are interested in obtaining a trace of a simple query, displaying the internal structures when something interesting happens. The query is compiled and then the debugger is invoked.

```
rfe-p> spy
rfe-p> app([1],[2],X)

((MAIN (VARI X)) (IS (VARI 1) (INST (CNS 1 NIL)))
 (IS (VARI 2) (INST (CNS 2 NIL))) (APP (VARI 1) (VARI 2) (VARI X)))

((PROC MAIN/1 1 (INDEXING)
  (FUN1EVA (CUT-INFO NIL) (PERM)
   (TEMP ((VARI X) (3 (1) (3))) ((VARI 1) (4 NIL (1))) ((VARI 2) (2 NIL (2))))
   (CHUNK
    ((USRLIT (MAIN ((VARI X) (FIRST SAFE TEMP))) (1 0 (1)))
     (IS ((VARI 1) (FIRST UNSAFE TEMP)) (INST (CNS 1 NIL)))
     (IS ((VARI 2) (FIRST UNSAFE TEMP)) (INST (CNS 2 NIL)))
     (USRLIT
      (APP ((VARI 1) (NONFIRST UNSAFE TEMP)) ((VARI 2) (NONFIRST UNSAFE TEMP))
       ((VARI X) (NONFIRST SAFE TEMP)))
      (3 0 (1 3))))
    (4 NIL)))))

((GET_X_VARIABLE 3 1) (PUT_LIST 4) (UNIFY_CONSTANT 1) (UNIFY_NIL) (PUT_LIST 2)
 (UNIFY_CONSTANT 2) (UNIFY_NIL) (PUT_X_VALUE 4 1) (EXECUTE APP/3))
```

The following is a debugger trace.

```
 [260932] = (GWAM.TRY 260934 0) : v

 Value of? s

 [160930] = unused-stack-cell <== E <== B
```

Initially there is not much on the stack. Registers E and B point to the beginning of the stack. The next instruction creates a choicepoint and the registers are set appropriately. This is the standard choicepoint which is responsible for the output of unknown/success messages, having the next clause entry pointing to code causing the output of the user's variables.

```
 [260932] = (GWAM.TRY 260934 0) : s

 [260934] = (GWAM.CALL/DY QUERY@[30514] 0) : v

 Value of? s

 [160930] = unused-stack-cell <== E
 [160931] = (ref 160930)
 [160932] = 260935
 [160933] = (ref 160930)
```

```
[160934] = 260933
[160935] = (trail nil)
[160936] = (ref 60931) <== B
[260934] = (GWAM.CALL/DY QUERY@[30514] 0) : s
[264018] = (GWAM.GET_X_VARIABLE 3 1) : s
[264019] = (GWAM.PUT_LIST 4) : s
[264020] = (GWAM.UNIFY_CONSTANT 1) : s
[264021] = (GWAM.UNIFY_NIL) : s
[264022] = (GWAM.PUT_LIST 2) : s
[264023] = (GWAM.UNIFY_CONSTANT 2) : s
[264024] = (GWAM.UNIFY_NIL) : s
[264025] = (GWAM.PUT_X_VALUE 4 1) : s
[264026] = (GWAM.EXECUTE/DY APP/3@[23842]) : v

Value of? a

Number of argument registers: 3

A(1) = (LIST 60932)
A(2) = (LIST 60934)
A(3) = (REF 60931)
[264026] = (GWAM.EXECUTE/DY APP/3@[23842]) : v

Value of? h

[60930] = unused-heap-cell <== S
[60931] = (ref 60931) <== HB
[60932] = (const 1)
[60933] = (const nil)
[60934] = (const 2)
[60935] = (const nil) <== H
[264026] = (GWAM.EXECUTE/DY APP/3@[23842]) : s
```

The code above allocates the structures for the query in the data space and sets the argument registers accordingly. Register X1 points to a list at memory locations 2 and 3, representing the list `(1 . nil)`, and register X2 points to the list at memory locations 4 and 5. The third argument (X3) is a reference to memory location 1, whose contents points to the same location. This is the representation of a free variable.

```
[263895] = (GWAM.SET_INDEX_NUMBER 1) : s
[263896] = (GWAM.SWITCH_ON_TERM 260931 260931 263905 263901 263897) : s
[263905] = (GWAM.ALLOCATE 0) : s
[263906] = (GWAM.GET_LIST 3) : s
```

Note that indexing leads the program flow immediately to the second clause of append/3.

```
[263907] = (GWAM.UNIFY_X_VARIABLE 4) : v

Value of? s

[160930] = unused-stack-cell
[160931] = (ref 160930)
[160932] = 260935
[160933] = (ref 160930)
[160934] = 260933
[160935] = (trail nil)
[160936] = (ref 60931) <== E <== B
[160937] = (ref 160930)
[160938] = 260935
[160939] = unused-stack-cell
[263908] = (GWAM.UNIFY_X_VARIABLE 5) : s
[263909] = (GWAM.GET_LIST 1) : s
[263910] = (GWAM.UNIFY_X_VALUE 4) : s
[263911] = (GWAM.UNIFY_X_VARIABLE 6) : s
[263912] = (GWAM.PUT_X_VALUE 6 1) : s
[263913] = (GWAM.PUT_X_VALUE 5 3) : s
[263914] = (GWAM.CALL/DY APP/3@[23842] 0) : v

Value of? a

Number of argument registers: 3

A(1) = (CONST NIL)
A(2) = (LIST 60934)
A(3) = (REF 60937)
[263914] = (GWAM.CALL/DY APP/3@[23842] 0) : v

[60930] = unused-heap-cell
[60931] = (list 60936) <== HB
[60932] = (const 1)
[60933] = (const nil) <== S
[60934] = (const 2)
[60935] = (const nil)
[60936] = (const 1)
[60937] = (ref 60937) <== H
[263914] = (GWAM.CALL/DY APP/3@[23842] 0) : s
```

Now app/3 is called with the following arguments: X1 is nil, X2 is (2.nil) and X3 is a free variable. Clearly, the first clause of app/3 must be applied.

```
[263895] = (GWAM.SET_INDEX_NUMBER 1) : s
[263896] = (GWAM.SWITCH_ON_TERM 260931 260931 263905 263901 263897) : s
[263901] = (GWAM.GET_NIL 1) : s
```

```
 [263902] = (GWAM.GET_X_VALUE 2 3) : s
 [263903] = (GWAM.PUT_CONSTANT TRUE 1) : s
 [263904] = (GWAM.PROCEED) : s
 [263915] = (GWAM.PUT_CONSTANT TRUE 1) : s
 [263916] = (GWAM.DEALLOCATE) : s
 [263917] = (GWAM.PROCEED) : s
 [260935] = (GWAM.HAS-SUCCEEDED) : v

 Value of? s


 [160930] = unused-stack-cell <== E
 [160931] = (ref 160930)
 [160932] = 263915
 [160933] = (ref 160930)
 [160934] = 260933
 [160935] = (trail nil)
 [160936] = (ref 60931) <== B
 [260935] = (GWAM.HAS-SUCCEEDED) : s

true
X=[1,2]
rfe-p> more
```

Indexing has pruned the search space for backtracking so that after the user's more request no other possibilities need be tested and the unknown message is generated.

```
 [260933] = (GWAM.TRUST 260930 0) : s
 [260930] = (GWAM.HAS-FAILED) : s

unknown
rfe-p>
```

References

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

[BAE+96] Harold Boley, Simone Andel, Klaus Elsbernd, Michael Herfert, Michael Sintek, and Werner Stein. RELFUN Guide: Programming with Relations and Functions Made Easy. Document D-93-12, DFKI GmbH, July 1996. Second, Revised Edition.

[Bol90] Harold Boley. A relational/functional Language and its Compilation into the WAM. SEKI Report SR-90-05, Universität Kaiserslautern, 1990.

[Bol92] Harold Boley. Extended Logic-plus-Functional Programming. In Lars-Henrik Eriksson, Lars Hallnäs, and Peter Schroeder-Heister, editors, *Proceedings of the 2nd International Workshop on Extensions of Logic Programming, ELP '91, Stockholm 1991*, volume 596 of *LNAI.* Springer, 1992.

[Els90] Klaus Elsbernd. Effizienzvergleiche zwischen einer LISP- und C-codierten WAM. SEKI Working Paper SWP-90-03, Universität Kaiserslautern, Fachbereich Informatik, June 1990.

[Hei89] Hans-Günther Hein. Adding WAM-Instructions to Support Valued Clauses for the Relational/Functional Language RELFUN. SEKI Working Paper SWP-90-02, Universität Kaiserslautern, Fachbereich Informatik, December 1989.

[Hei91] Hans-Günther Hein. WAM indexing and footening techniques for RELFUN — a case study on the DNF benchmark. ARC-TEC Discussion Paper 91-11, DFKI Kaiserslautern, August 1991.

[Her92] Michael Herfert. Parsen und Generieren der PROLOG-artigen Syntax von RELFUN. Technical Report D-92-23, DFKI GmbH, October 1992.

[Kra90] Thomas Krause. Klassifizierte relational/funktionale Klauseln: Eine deklarative Zwischensprache zur Generierung von Register-optimierten WAM-Instruktionen. SEKI Working Paper SWP-90-04, Universität Kaiserslautern, Fachbereich Informatik, May 1990.

[Kra91] Thomas Krause. Globale Datenflußanalyse und horizontale Compilation der relational-funktionalen Sprache RELFUN. Diplomarbeit, DFKI D-91-08, Universität Kaiserslautern, FB Informatik, Postfach 3049, D-6750 Kaiserslautern, March 1991.

[Nys] Sven Olof Nystrøm. Nywam - a WAM emulator written in LISP.

[Per96] Markus Perling. RAWAM - a Relfun Adapted WAM, 1996.

[Sin93] Michael Sintek. Indexing PROLOG procedures into DAGs by heuristic classification. DFKI Technical Memo TM-93-05, DFKI GmbH, 1993.

[Sin95] Michael Sintek. FLIP: Functional-plus-logic programming on an integrated platform. Technical Memo TM-95-02, DFKI GmbH, May 1995.

[SS92] Werner Stein and Michael Sintek. A generalized intelligent indexing method. In *Workshop "Sprachen für KI-Anwendungen, Konzepte - Methoden - Implementierungen" in Bad Honnef, 12/92-1*. Institute of Applied Mathematics and Computer Science, University of Münster, May 1992.

[Ste93] Werner Stein. Indexing Principles for Relational Languages Applied to PROLOG Code Generation. Technical Report Document D-92-22, DFKI GmbH, February 1993.

[VR94] Peter Van Roy. 1983-1993: The wonder years of sequential Prolog implementation. *The Journal of Logic Programming*, 19,20:385–441, 1994.

[War83] David. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Menlo Park, CA, October 1983.

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

-Bibliothek, Information
und Dokumentation (BID)-
PF 2080
67608 Kaiserslautern
FRG

Telefon (0631) 205-3506
Telefax (0631) 205-3210
e-mail
dfkibib@dfki.uni-kl.de
WWW
http://www.dfki.uni-sb.de/dfkibib

Veröffentlichungen des DFKI

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse oder (so sie als per ftp erhaeltlich angemerkt sind) per anonymous ftp von ftp.dfki.uni-kl.de (131.246.241.100) im Verzeichnis pub/Publications bezogen werden. Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of all published papers so far are obtainable from the above address or (if they are marked as obtainable by ftp) by anonymous ftp from ftp.dfki.uni-kl.de (131.246.241.100) in the directory pub/Publications.
The reports are distributed free of charge except where otherwise noted.

DFKI Research Reports

1996

RR-96-05
Stephan Busemann
Best-First Surface Realization
11 pages

RR-96-03
Günter Neumann
Interleaving
Natural Language Parsing and Generation
Through Uniform Processing
51 pages

RR-96-02
E.André, J. Müller , T.Rist:
PPP-Persona: Ein objektorientierter Multimedia-Präsentationsagent
14 Seiten

1995

RR-95-20
Hans-Ulrich Krieger
Typed Feature Structures, Definite Equivalences, Greatest Model Semantics, and Nonmonotonicity
27 pages

RR-95-19
Abdel Kader Diagne, Walter Kasper, Hans-Ulrich Krieger
Distributed Parsing With HPSG Grammar
20 pages

RR-95-18
Hans-Ulrich Krieger, Ulrich Schäfer
Efficient Parameterizable Type Expansion for Typed Feature Formalisms
19 pages

RR-95-17
Hans-Ulrich Krieger
Classification and Representation of Types in TDL
17 pages

RR-95-16
Martin Müller, Tobias Van Roy
Title not set
0 pages

Note: The author(s) were unable to deliver this document for printing before the end of the year. It will be printed next year.

RR-95-15
Joachim Niehren, Tobias Van Roy
Title not set
0 pages

Note: The author(s) were unable to deliver this document for printing before the end of the year. It will be printed next year.

RR-95-14
Joachim Niehren
Functional Computation as Concurrent Computation
50 pages

RR-95-13
Werner Stephan, Susanne Biundo
Deduction-based Refinement Planning
14 pages

RR-95-12
Walter Hower, Winfried H. Graf
Research in Constraint-Based Layout, Visualization, CAD, and Related Topics: A Bibliographical Survey
33 pages

RR-95-11
Anne Kilger, Wolgang Finkler
Incremental Generation for Real-Time Applications
47 pages

RR-95-10
Gert Smolka
The Oz Programming Model
23 pages

RR-95-09
M. Buchheit, F. M. Donini, W. Nutt, A. Schaerf
A Refined Architecture for Terminological Systems: Terminology = Schema + Views
71 pages

RR-95-08
Michael Mehl, Ralf Scheidhauer, Christian Schulte
An Abstract Machine for Oz
23 pages

RR-95-07
Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, Werner Nutt
The Complexity of Concept Languages
57 pages

RR-95-06
Bernd Kiefer, Thomas Fettig
FEGRAMED
An interactive Graphics Editor for Feature Structures
37 pages

RR-95-05
Rolf Backofen, James Rogers, K. Vijay-Shanker
A First-Order Axiomatization of the Theory of Finite Trees
35 pages

RR-95-04
M. Buchheit, H.-J. Bürckert, B. Hollunder, A. Laux, W. Nutt,
M. Wójcik
Task Acquisition with a Description Logic Reasoner
17 pages

RR-95-03
Stephan Baumann, Michael Malburg, Hans-Guenther Hein, Rainer Hoch,
Thomas Kieninger, Norbert Kuhn
Document Analysis at DFKI
Part 2: Information Extraction
40 pages

RR-95-02
Majdi Ben Hadj Ali, Frank Fein, Frank Hoenes, Thorsten Jaeger,
Achim Weigel
Document Analysis at DFKI
Part 1: Image Analysis and Text Recognition
69 pages

RR-95-01
Klaus Fischer, Jörg P. Müller, Markus Pischel
Cooperative Transportation Scheduling
an application Domain for DAI
31 pages

1994

RR-94-39
Hans-Ulrich Krieger
Typed Feature Formalisms as a Common Basis for Linguistic Specification.
21 pages

RR-94-38
Hans Uszkoreit, Rolf Backofen, Stephan Busemann, Abdel Kader Diagne,
Elizabeth A. Hinkelman, Walter Kasper, Bernd Kiefer, Hans-Ulrich Krieger,
Klaus Netter, Günter Neumann, Stephan Oepen, Stephen P. Spackman.
DISCO–An HPSG-based NLP System and its Application for Appointment Scheduling.
13 pages

RR-94-37
Hans-Ulrich Krieger, Ulrich Schäfer
TDL - A Type Description Language for HPSG, Part 1: Overview.
54 pages

RR-94-36
Manfred Meyer
Issues in Concurrent Knowledge Engineering. Knowledge Base and Knowledge Share Evolution.
17 pages

RR-94-35
Rolf Backofen
A Complete Axiomatization of a Theory with Feature and Arity Constraints
49 pages

RR-94-34
Stephan Busemann, Stephan Oepen, Elizabeth A. Hinkelman,
Günter Neumann, Hans Uszkoreit
COSMA – Multi-Participant NL Interaction for Appointment Scheduling
80 pages

RR-94-33
Franz Baader, Armin Laux
Terminological Logics with Modal Operators
29 pages

RR-94-31
Otto Kühn, Volker Becker, Georg Lohse, Philipp Neumann
Integrated Knowledge Utilization and Evolution for the Conservation of Corporate Know-How
17 pages

RR-94-23
Gert Smolka
The Definition of Kernel Oz
53 pages

RR-94-20
Christian Schulte, Gert Smolka, Jörg Würtz
Encapsulated Search and Constraint Programming in Oz
21 pages

RR-94-19
Rainer Hoch
Using IR Techniques for Text Classification in Document Analysis
16 pages

RR-94-18
Rolf Backofen, Ralf Treinen
How to Win a Game with Features
18 pages

RR-94-17
Georg Struth
Philosophical Logics—A Survey and a Bibliography
58 pages

RR-94-16
Gert Smolka
A Foundation for Higher-order Concurrent Constraint Programming
26 pages

RR-94-15
Winfried H. Graf, Stefan Neurohr
Using Graphical Style and Visibility Constraints for a Meaningful Layout in Visual Programming Interfaces
20 pages

RR-94-14
Harold Boley, Ulrich Buhrmann, Christof Kremer
Towards a Sharable Knowledge Base on Recyclable Plastics
14 pages

RR-94-13
Jana Koehler
Planning from Second Principles—A Logic-based Approach
49 pages

RR-94-12
Hubert Comon, Ralf Treinen
Ordering Constraints on Trees
34 pages

RR-94-11
Knut Hinkelmann
A Consequence Finding Approach for Feature Recognition in CAPP
18 pages

RR-94-10
Knut Hinkelmann, Helge Hintze
Computing Cost Estimates for Proof Strategies
22 pages

RR-94-08
Otto Kühn, Björn Höfling
Conserving Corporate Knowledge for Crankshaft Design
17 pages

RR-94-07
Harold Boley
Finite Domains and Exclusions as First-Class Citizens
25 pages

RR-94-06
Dietmar Dengler
An Adaptive Deductive Planning System
17 pages

RR-94-05
Franz Schmalhofer, J. Stuart Aitken, Lyle E. Bourne jr.
Beyond the Knowledge Level: Descriptions of Rational Behavior for Sharing and Reuse
81 pages

RR-94-03
Gert Smolka
A Calculus for Higher-Order Concurrent Constraint Programming with Deep Guards
34 pages

RR-94-02
Elisabeth André, Thomas Rist
Von Textgeneratoren zu Intellimedia-Präsentationssystemen
22 Seiten

RR-94-01
Elisabeth André, Thomas Rist
Multimedia Presentations: The Support of Passive and Active Viewing
15 pages

DFKI Technical Memos

1996

TM-96-01
Gerd Kamp, Holger Wache
CTL — a description Logic with expressive concrete domains
19 pages

1995

TM-95-04
Klaus Schmid
Creative Problem Solving
and
Automated Discovery
— An Analysis of Psychological and AI Research —
152 pages

TM-95-03
Andreas Abecker, Harold Boley, Knut Hinkelmann, Holger Wache,
Franz Schmalhofer
An Environment for Exploring and Validating Declarative Knowledge
11 pages

TM-95-02
Michael Sintek
FLIP: Functional-plus-Logic Programming
on an Integrated Platform
106 pages

TM-95-01
Martin Buchheit, Rüdiger Klein, Werner Nutt
Constructive Problem Solving: A Model Construction Approach towards Configuration
34 pages

1994

TM-94-04
Cornelia Fischer
PAntUDE - An Anti-Unification Algorithm for Expressing Refined Generalizations
22 pages

TM-94-03
Victoria Hall
Uncertainty-Valued Horn Clauses
31 pages

TM-94-02
Rainer Bleisinger, Berthold Kröll
Representation of Non-Convex Time Intervals and Propagation of Non-Convex Relations
11 pages

TM-94-01
Rainer Bleisinger, Klaus-Peter Gores
Text Skimming as a Part in Paper Document Understanding
14 pages

DFKI Documents

1996

D-96-05
Martin Schaaf
Ein Framework zur Erstellung verteilter Anwendungen
94 pages

D-96-03
Winfried Tautges
Der DESIGN-ANALYZER - Decision Support im Designprozess
75 Seiten

1995

D-95-12
F. Baader, M. Buchheit, M. A. Jeusfeld, W. Nutt (Eds.)
Working Notes of the KI'95 Workshop:
KRDB-95 - Reasoning about Structured Objects:
Knowledge Representation Meets Databases
61 pages

D-95-11
Stephan Busemann, Iris Merget
Eine Untersuchung kommerzieller Terminverwaltungssoftware im Hinblick auf die Kopplung mit natürlichsprachlichen Systemen
32 Seiten

D-95-10
Volker Ehresmann
Integration ressourcen-orientierter Techniken in das wissensbasierte Konfigurierungssystem TOOCON
108 Seiten

D-95-09
Antonio Krüger
PROXIMA: Ein System zur Generierung graphischer Abstraktionen
120 Seiten

D-95-08
Technical Staff
DFKI Jahresbericht 1994
63 Seiten

Note: This document is no longer available in printed form.

D-95-07
Ottmar Lutzy
Morphic - Plus
Ein morphologisches Analyseprogramm für die deutsche Flexionsmorphologie und Komposita-Analyse
74 pages

D-95-06
Markus Steffens, Ansgar Bernardi
Integriertes Produktmodell für Behälter aus Faserverbundwerkstoffen
48 Seiten

D-95-05
Georg Schneider
Eine Werkbank zur Erzeugung von 3D-Illustrationen
157 Seiten

D-95-04
Victoria Hall
Integration von Sorten als ausgezeichnete taxonomische Prädikate in eine relational-funktionale Sprache
56 Seiten

D-95-03
Christoph Endres, Lars Klein, Markus Meyer
Implementierung und Erweiterung der Sprache $\mathcal{ALCP}$
110 Seiten

D-95-02
Andreas Butz
BETTY
Ein System zur Planung und Generierung informativer Animationssequenzen
95 Seiten

D-95-01
Susanne Biundo, Wolfgang Tank (Hrsg.)
PuK-95, Beiträge zum 9. Workshop „Planen und Konfigurieren“, Februar 1995
169 Seiten

Note: This document is available for a nominal charge of 25 DM (or 15 US-$).

1994

D-94-15
Stephan Oepen
German Nominal Syntax in HPSG
— On Syntactic Categories and Syntagmatic Relations
—
80 pages

D-94-14
Hans-Ulrich Krieger, Ulrich Schäfer
TDL - A Type Description Language for HPSG, Part 2: User Guide.
72 pages

D-94-12
Arthur Sehn, Serge Autexier (Hrsg.)
Proceedings des Studentenprogramms der 18. Deutschen Jahrestagung für Künstliche Intelligenz KI-94
69 Seiten

D-94-11
F. Baader, M. Buchheit, M. A. Jeusfeld, W. Nutt (Eds.)
Working Notes of the KI'94 Workshop: KRDB'94 - Reasoning about Structured Objects: Knowledge Representation Meets Databases
65 pages

Note: This document is no longer available in printed form.

D-94-10
F. Baader, M. Lenzerini, W. Nutt, P. F. Patel-Schneider (Eds.)
Working Notes of the 1994 International Workshop on Description Logics
118 pages

Note: This document is available for a nominal charge of 25 DM (or 15 US-$).

D-94-09
Technical Staff
DFKI Wissenschaftlich-Technischer Jahresbericht 1993
145 Seiten

D-94-08
Harald Feibel
IGLOO 1.0 - Eine grafikunterstützte Beweisentwicklungsumgebung
58 Seiten

D-94-07
Claudia Wenzel, Rainer Hoch
Eine Übersicht über Information Retrieval (IR) und NLP-Verfahren zur Klassifikation von Texten
25 Seiten

D-94-06
Ulrich Buhrmann
Erstellung einer deklarativen Wissensbasis über recyclingrelevante Materialien
117 Seiten

D-94-04
Franz Schmalhofer, Ludger van Elst
Entwicklung von Expertensystemen: Prototypen, Tiefenmodellierung und kooperative Wissensevolution
22 Seiten

D-94-03
Franz Schmalhofer
Maschinelles Lernen: Eine kognitionswissenschaftliche Betrachtung
54 Seiten

Note: This document is no longer available in printed form.

D-94-02
Markus Steffens
Wissenserhebung und Analyse zum Entwicklungsprozeß eines Druckbehälters aus Faserverbundstoff
90 pages

D-94-01
Josua Boon (Ed.)
DFKI-Publications: The First Four Years 1990 - 1993
75 pages

RFM Manual: Compiling RELFUN into the Relational/Functional Machine

(Third, Revised Edition)

Harold Boley, Klaus Elsbernd, Hans-Günther Hein, Thomas Krause,
Markus Perling, Michael Sintek, Werner Stein

D-91-03
Document