FLIP:
Functional-plus-Logic Programming on an Integrated Platform

Michael Sintek
DFKI GmbH
Postfach 2080
67608 Kaiserslautern
Germany

May 1995

Abstract

In this work, a novel approach to the integration of relational and functional languages on the basis of abstract machines (in the context of the RELFUN language and implementation) is described.

This integration is carried out for several reasons: to combine two declarative paradigms into a more expressive one, to allow existing software libraries in relational and functional (here LL, a COMMON LISP derivative) languages to be used together without the need of re-implementation, to speed up relational programs by transforming deterministic relations into functions, and to enhance the expressiveness of relational languages by new extra-logicals with the help of functions.

The integration is performed on two levels: 1. on the abstract machine level (the WAM, the abstract machine behind most implementations of relational languages, and the LLAMA, an abstract machine especially designed for the efficient execution of LL, are coupled), and 2. on the source language level (LL functions are accessible from relations and vice versa).

One of the major points of this work is the detection and transformation of deterministic relations (into LL functions), resulting in a speed-up factor of 2-4. For this, a theoretical foundation for deterministic relations and several intermediate representation languages for the transformation process are developed.

Contents

Chapter 1

Introduction

1.1 Objectives

The objective of this work is a novel approach to the integration of functional and logic programming languages on the basis of abstract machines in the context of the RELFUN [Boley, 1992] project. This integration is motivated by the following points:

1. Both functional and logic programming are declarative paradigms, but are suitable for solving partially disjoint problem classes: logic programming is preferable for problems involving *search*, functional programming is preferable for *deterministic symbolic computations* on complex dynamic data structures.

   In real-world applications, often problems of both problem classes occur, thus solving them in a single programming paradigm requires some abuse of it. As a consequence, large programs developed in integrated functional-logic programming languages profit from the advantages both languages can offer.

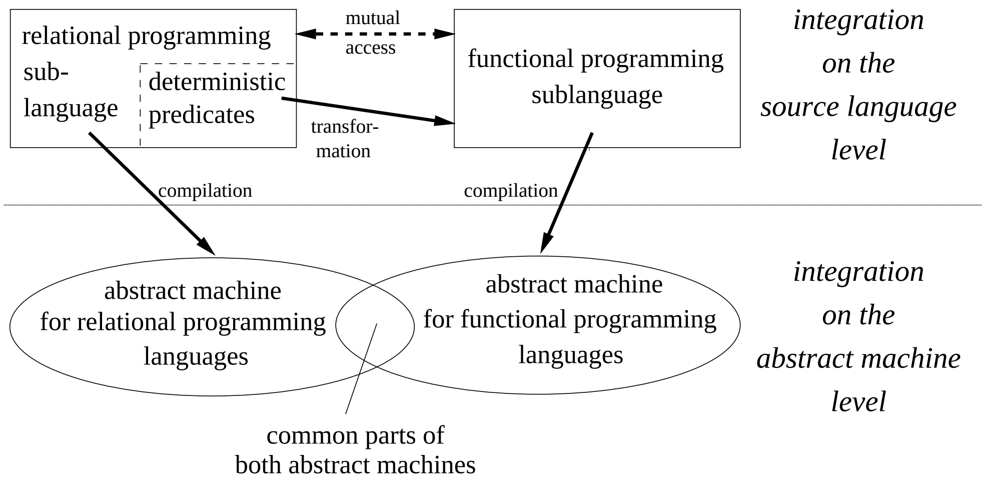
2. In addition to the development of new algorithms, reuse of existing software libraries plays an important role in practice, since software development and especially software maintenance is highly expensive. For this reason, stable software should be reused as often as possible, even if the programming language it has been developed in turns out to be suboptimal in some sense ("Never change a running system."). It is thus desirable to permit combining already existing algorithms of which some are specified in a functional and some in a logic programming language, without having to re-implement any of them in the other paradigm.

3. Logic programming languages usually have the disadvantage of being inefficient since unification and search are not directly supported by von Neumann architectures. Since the development of hardware directly suited for

logic programming languages turned out to proceed slower than expected [Benker *et al.*, 1989; Dorochevsky *et al.*, 1991; Kurozumi, 1992], other means for improving the efficiency of logic programs have to be developed, too. In addition to recent research in the field of *native code generation* for logic programs [Taylor, 1990; Van Roy, 1994], research dealing with the improved compilation of logic programs into *abstract machines* should be pursued, too.

In the current work, the integration of logic and functional programming with abstract machines will be shown to be suitable for attacking the efficiency problem for the class of logic programs containing predicates that are only used deterministically: in this case, these deterministic predicates are transformed into functions which are then compiled into an abstract machine especially designed for the efficient execution of functional languages. By this, the overhead caused by the general search strategies — wastefully applied to deterministic predicates — can be avoided.

The following diagram illustrates the *integration* of logic and functional programming on the basis of abstract machines and the *transformation* of deterministic predicates into the functional sublanguage:



4. In concrete logic programming languages, it is often desirable to enhance the expressiveness by adding new features. This is usually either performed by directly extending (changing) the compiler/interpreter or by designing the language for extensibility via (usually extra-logical) programs written in the language itself.[1]

[1] For example, some logic programming languages allow the unification, constraint solving, and the search strategies to be extended with the help of programs written in the language itself.

While the first approach is rather inflexible (i.e. expensive, error-prone, non-modular), the second is usually inefficient and often introduces performance penalties even for programs not requiring any extensions. In this work, a flexible and efficient means for extending the expressiveness of logic programs with the help of deterministic functions will be shown.

1.2 Approach and Results

In this work, the integration of

1. relational languages (with REL as a representative) and
2. LL (LISP *light*), a subset of COMMON LISP extended with PROLOG-like structures,

is presented. The extension of LL with PROLOG-like structures is necessary in order to permit both sublanguages to work on the same data types.

This integration is carried out on two levels:

- *On the abstract machine level:*
  The WAM [Warren, 1983], the abstract machine which has turned out to be a (local) optimum for the compilation of relational languages, is (loosely) coupled with the LLAMA (LISP *light* Abstract Machine), an abstract stack machine especially designed for the efficient execution of LL programs while handling all PROLOG data structures (as they are internally represented) to avoid their transformation at run time.

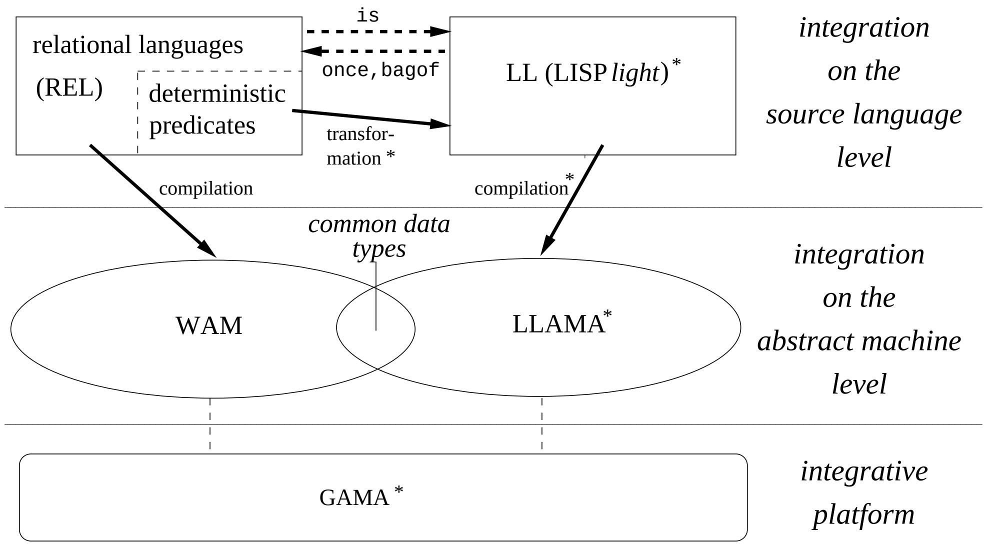
  The integration of the WAM and the LLAMA was carried out with the help of the GAMA, a General Abstract Machine Assembler, intended as a programming environment supporting the development and integration of abstract machines.

- *On the source language level:*
  1. As a result of the integration of the two underlying abstract machines, relational programs compiled into the WAM and LL programs compiled into the LLAMA can access each other:[2]
     relations access LL functions via a generalized `is` builtin, allowing not only arithmetical builtins (as in PROLOG) but also LL functions on the right-hand side, and LL functions access relations, either with `once` (retrieving only the first solution), or with a `bagof`-like construct (retrieving all solutions).

[2] In our prototypical implementation, only relational programs access LL programs.

2. Relations that are (detected or declared to be) only used deterministically are transformed into LL functions and then compiled into LLAMA code, thus avoiding the overhead caused by the ordinary WAM code.

The following diagram, a refinement of the diagram on page 5, illustrates the integration of relational languages and LL on both the source language and the abstract machine level (parts newly developed in this work are marked with *):



This approach meets all the requirements presented in the preceding section:

1. Applications in which either the logic or the functional programming paradigm is best suited for some sub-problems can be developed in a relational language coupled with LL.

2. Already existing programs in relational languages and LISP can be integrated without re-implementation. Since PROLOG and COMMON LISP are the predominant logic and functional programming languages, mainly subsets of these languages have been chosen.

3. By mapping deterministic predicates into LL functions, the efficiency of relational programs can often be considerably improved (by a factor of 2-4; see appendix A). Especially, large relational applications tend to contain a huge number of deterministic predicates, as we were able to determine via some of our RELFUN programs [Sintek, 1991; Boley, 1991].

4. The expressiveness of relational programs can be improved by specifying new features as LL functions and using them as pseudo builtins.

Compared with more theoretically motivated approaches using a *tight* integration of logic and functional languages on the source and the abstract machine levels (e.g. [Hanus, 1991; Lock, 1993]), our approach has the following strong points:

- Already existing programs in PROLOG and LISP do not have to be re-implemented in a new, unified formalism.
- For applications requiring additional programming paradigms, further abstract machines can be coupled with the WAM and the LLAMA without much effort.
- The loose coupling of the WAM and the LLAMA allows new insights in the technology of the WAM or stack machines to be easily adopted.

1.3 Outline

Chapter 2 presents the **background** for this work: a short introduction is given for relational-functional programming languages and abstract machines.

Chapter 3 introduces **LL**: its data types, special forms and builtin functions, and the user-definable functions. Since LL is mainly a subset of COMMON LISP, the reader familiar with COMMON LISP may skip most of the details in this chapter.

Chapter 4 describes the **integration of relational languages and LL**: their integration on the abstract machine and source language level, the transformation of deterministic predicates into LL functions, and the implementation of extra-logicals with the help of LL. Since the transformation of deterministic predicates requires their detection, a theoretical foundation for deterministic relations (using SLD resolution trees) is developed.

Chapter 5 gives a description of the **LLAMA**: its registers and memory organization, the representation of data structures, the calling conventions, and the instructions. The LLAMA is presented in a rather technical manner using machine state transition diagrams, thus allowing the LLAMA to be easily re-implemented in any (imperative or functional) programming language.

Chapter 6 describes the **compiler for LL into the LLAMA**. Again, a technical representation using transformation rules was employed in order to allow the compiler, just like the LLAMA, to be easily re-implemented.

Chapter 7 gives a short overview of the **GAMA** which was used to integrate the WAM with the LLAMA.

Chapter 8 finishes this work with **conclusions** and gives some hints for **future work**.

Chapter 2

Background

In this chapter, a short introduction to relational-functional programming and abstract machines is presented.

The reader is expected to be familiar with the concepts of logic programming and functional programming. In [Lloyd, 1987], an introduction to logic programming, in [Wikstrøm, 1987], an introduction to functional programming is given.

2.1 Relational-Functional Programming Languages

Languages integrating functional and logic (relational) programming concepts [DeGroot and Lindstrom, 1986] are currently often called *functional logic* programming languages. However, we will employ the term *relational-functional* programming languages, where the term *relational language* is used as a superconcept for both *purely logic languages* and *logic languages enhanced by (functionally appropriate but) extra-logical features* such as the cut/commit operator.

The integration of different programming paradigms into a new programming language aims at a language inheriting the advantages of the original languages. In the case of the integration of relational and functional programming languages, their declarativeness and, in particular, their expressiveness, i.e. their suitability for partially disjoint problem classes, is inherited.

In general, there are two different ways of integrating functional and relational programming languages [Boley, 1992]:

1. by *tightly integrating* both paradigms into unified ones [Boley, 1986; Hanus, 1991; Lock, 1993]

2. by *loose coupling*, obtaining hybrid languages (for classic systems, see [Sloman and Hardy, 1983; Robinson, 1985]) which are often parts of multi-language expert system shells

Both approaches have their advantages and disadvantages:

1. While tightly integrated languages maintain full declarativeness, they require existing software libraries to be re-implemented and cannot be easily augmented by additional programming paradigms. Furthermore, tightly integrated languages often lose some of the efficiency of the separate languages since techniques of their efficient execution can usually not be entirely transferred to the integrated language.

2. Hybrid languages have the disadvantage of not completely maintaining declarativeness: applications have to be split into parts specified in different paradigms and communicating via interface primitives. For large, modular applications this is often insignificant: complete modules can be specified either in the functional *or* the relational sublanguage, accessing each other only via well-defined interfaces.[1]

The loose integration of relational languages and LISP in this work was inspired by RELFUN [Boley, 1992], a tightly integrated <u>rel</u>ational-plus-<u>fun</u>ctional programming language also building upon PROLOG and LISP (the original version [Boley, 1986] not only used a LISP-like syntax for both relations and functions but also LISP-like dotted-pair patterns for unification).

In addition to functions being accessible on the right-hand side of the generalized `is` builtin, RELFUN also allows function calls in each premise, all of which can be nested for call-by-value evaluation. This requires a syntactical distinction between *active* function applications and *passive* structures: while round parentheses are reserved for applications, the square brackets used for PROLOG lists are generalized to be also used for structures (which can thus be viewed as labeled lists).

In RELFUN, the tight integration of functions into a relational language is performed via functional clauses, an extension of relational (Horn) clauses by a value-returning premise (the last premise, preceded by an ampersand, "`&`"). This directly leads to the main difference between RELFUN and the loose integration described in this work: RELFUN functions inherit all properties of ordinary relations and are thus potentially *non-deterministic*, as the following example illustrates:

Example 2.1
The *ternary* PROLOG relation

```
memberp(X, [X | R], [X | R]).
memberp(X, [_ | R], Z) :- member(X, R, Z).
```

[1] In the case of applications developed by more than one implementor, the languages for the separate modules can eventually be chosen according to their preferences or abilities.

can in RELFUN be rewritten as the *binary* (LISP-inspired) function

```
member(X, [X | R]) :-& [X | R].
member(X, [_ | R]) :-& member(X, R).
```

For lists containing the searched element more than once, each occurrence results in a separately returned solution:

```
rfe-p> member(1, [0,1,2,3,1,4,6])
[1, 2, 3, 1, 4, 6]
rfe-p> more
[1, 4, 6]
rfe-p> more
unknown
```

△

For relations and functions to be applied only deterministically, RELFUN extends PROLOG's cut use to its functional clauses:[2]

```
memberp(X, [X | R], [X | R]) :- !.
memberp(X, [_ | R], Z) :- member(X, R, Z).

member(X, [X | R]) :- ! & [X | R].
member(X, [_ | R]) :-& member(X, R).
```

Now, `member(1, [0,1,2,3,1,4,6])` succeeds only once (with the first solution).

A principal goal of this work will be to transform the deterministic `memberp` relation into the deterministic `member` function and both into the LISP-like LL-language for execution on the highly efficient stack machine LLAMA.

For providing some notation, as our highest-level relational (input) language we will use a sublanguage of RELFUN, here called REL, encompassing relations, the generalized `is`-primitive, as well as cut and `once` operators.

REL clauses will thus employ RELFUN's syntax, e.g. structures will be enclosed in square brackets, since this generalization is useful in the context of relational-functional languages (and can even enhance the readability of purely relational languages). Furthermore, RELFUN functions will be used as one of our intermediate representation languages in the transformation of REL into LL (section 4.2.3).

[2]Since in RELFUN these deterministic clauses are compiled into ordinary WAM code (in order to allow unification and non-ground computations in functions), efficiency is not much improved.

2.2 Abstract Machines

The execution of programming languages in general can be performed in three different ways:

1. by *interpreting* the source code,
2. by *compiling* the source code into *native machine code*, or
3. by *compiling* it into an *abstract machine code* and then emulating it.

Interpretation is used for the definition of the operational semantics and in prototypical implementations of all kinds of programming languages. In order to obtain efficiency, compilation is needed. Programming languages complying with the imperative programming style imposed by the von Neumann architecture are usually compiled into native machine code.

For a language like PROLOG which cannot be easily mapped into the imperative programming paradigm, the compilation into an abstract machine especially designed for it is usually a good compromise w.r.t. efficiency and development costs. The machine code for such an abstract machine is interpreted by an ordinary (but of course highly optimized) imperative program in contrast to the execution of native machine code with a hardware-encoded interpreter (a processor).

In case of relational programming languages, for quite some time only interpreters existed. In 1967, Absys (Aberdeen System), an interpreter for (a predecessor of) pure PROLOG, was developed at the University of Aberdeen. In 1977, David H. D. Warren et. al. developed DEC-10 PROLOG, the first PROLOG compiler, and in 1983, he designed the WAM [Warren, 1983], the abstract machine that has become the de facto standard implementation technique for relational languages [Van Roy, 1994].

The WAM can be characterized as follows:

WAM =		sequential control	(call/return/jump instructions)
	+	unification	(get/put/unify instructions)
	+	backtracking	(try/retry/trust instructions)

Since the coupling of the WAM and the LLAMA is loose, a deep understanding of the WAM will not be required.[3] Only

- the WAM heap (holding lists and structures, the compound data terms of PROLOG and RELFUN/REL),
- the local stack (holding environments, which contain local variables, and choice points), and

[3] For introductions into the WAM, refer to [Gabriel *et al.*, 1985], [Aït-Kaci, 1990], and [Van Roy, 1994].

- some of the WAM registers (the program counter and the stack and heap pointers)

are shared between the WAM and the LLAMA. They will be described in detail in sections 5.1 and 5.2.

Chapter 3

LL: LISP light

LL (LISP *light*) is a subset of (COMMON) LISP [McCarthy *et al.*, 1962; Steele Jr., 1984] extended by PROLOG structures. In addition to purely functional builtins, it contains some extra-functional builtins for the following reasons:

1. Existing (COMMON) LISP programs can be *integrated* with relational programs without having to re-implement their non-functional parts like global variables, re-assignments via `setq`, or loops.

2. LL functions can be used to *augment* relational languages by (new) extra-logicals. Some of these extensions require non-functional behaviour, e.g. `bagof`, which can be implemented via global LL variables (see section 4.3).

3. Deterministic tail-recursive relations and functions can be translated into loops to *speed up* execution.

In the following, LL's data types, special forms, builtins, and user-definable functions will be described.

3.1 Data Types

In LL, the following data types[1] are supported:

- symbols
- numbers (integers and reals)
- strings
- lists and dotted pairs

[1] In the context of LISP, data types are usually referred to as S-expressions (symbolic expressions) [McCarthy *et al.*, 1962].

- structures

Symbols, numbers, strings, and lists work just like their COMMON LISP counterparts. Because LL has to handle all data types that can be found on the WAM stack and heap, PROLOG structures had to be added. They are quite similar to (1-dimensional) COMMON LISP arrays in that they allow efficient random access to all of their elements in contrast to LISP/PROLOG lists which require linear access by applying a `cdr` operation repeatedly.

In LL, structures are enclosed in square brackets and use LISP's Cambridge Polish prefix notation:

PROLOG	RELFUN	LL
`f(a,b,c)`	`f[a,b,c]`	`[f a b c]`

LL structures evaluate to themselves, thus quoting is not necessary but — as with numbers and strings — allowed.

In analogy to constructing lists with `list` or `cons`, LL structures can be built up with `struct` ($\stackrel{\varepsilon}{\rightarrow}$ denotes evaluation; $expr^{\varepsilon} = expr'$ is equivalent to $expr \stackrel{\varepsilon}{\rightarrow} expr'$):

$$\texttt{(struct 'f 'a 'b 'c)} \stackrel{\varepsilon}{\rightarrow} \texttt{[f a b c]}$$
$$\texttt{(struct 'f 'a (struct 'g 'b 'c))} \stackrel{\varepsilon}{\rightarrow} \texttt{[f a [g b c]]}$$

Using `struct` is the only way to construct a structure which contains variables. The RELFUN/REL goal

$$\texttt{X is a, Y is f[X,b]} \rightarrow \texttt{Y = f[a, b]}$$

has, in LL, to be expressed as:

$$\texttt{(let ((x 'a)) (struct 'f x 'b))} \stackrel{\varepsilon}{\rightarrow} \texttt{[f a b]}$$

LL — just like COMMON LISP — does not support pattern matching or unification. Selecting the constituents of a structure has to be done with the following functions:

- `(arity [`$f\ arg_1\ \ldots\ arg_n$`])` $\stackrel{\varepsilon}{\rightarrow} n$
- `(functor [`$f\ arg_1\ \ldots\ arg_n$`])` $\stackrel{\varepsilon}{\rightarrow} f$
- `(elt [`$f\ arg_0\ \ldots\ arg_n$`]` m`)` $\stackrel{\varepsilon}{\rightarrow} arg_m$ with $0 \le m \le n$

3.2 Special Forms and Builtin Functions

In LL, only basic special forms and builtin functions are directly understood by the compiler. Often-used builtins that can be defined via these basic special forms and builtins are contained in the prelude which is loaded at system startup.

In the following subsections, the LL builtins will be described. If the reader is familiar with COMMON LISP, most of these subsections may be skipped (except subsection 3.2.3, which describes LL structures).

Only short definitions of the LL builtins are given. A detailed description can be found in the LISP literature [Steele Jr., 1984].

LL Builtins	
quoting	`quote`, `'`
lists and dotted pairs	`cons`, `list`, `car`, `cdr`, `null`, `consp`
structures	`struct`, `structp`, `arity`, `functor`, `elt`
(single/re-) assignments	`let`, `let*`, `setq`, `psetq`
equality	`equal`, `eq`, `eql`
numerical builtins	`+`, `-`, `*`, `/`, `1+`, `1-`, `=`, `/=`, `<`, `>`, `<=`, `>=`
string builtins	`string<`, `string>`
I/O builtins	`read`, `print`
higher-order builtins	`funcall`, `apply`, `lambda`, `function`, `eval`
sequential evaluation	`progn`
conditional evaluation	`if`, `cond`, `and`, `or`
non-local exits and loops	`catch`, `throw`, `loop`, `return`, `do`
prelude	`caar`, `cadr`, `cdar`, `cddr`,
	`mapcar`, `append`, `reverse`, `member`, `assoc`, `sort`

3.2.1 Quoting

- $\texttt{(quote}\ arg\texttt{)} \stackrel{\varepsilon}{\to} arg$
- $\texttt{'}arg \stackrel{\varepsilon}{\to} arg$

3.2.2 Lists and Dotted Pairs

- $\texttt{(cons}\ a\ b\texttt{)} \stackrel{\varepsilon}{\to} \texttt{(}a^\varepsilon\ \texttt{.}\ b^\varepsilon\texttt{)}$
- $\texttt{(list}\ expr_1\ \ldots\ expr_n\texttt{)} \stackrel{\varepsilon}{\to} \texttt{(}expr_1^\varepsilon\ \ldots\ expr_n^\varepsilon\texttt{)}$
- $\texttt{(car}\ l\texttt{)} \stackrel{\varepsilon}{\to} \begin{cases} \texttt{nil} & , l^\varepsilon = \texttt{nil} \\ x & , l^\varepsilon = (x\ .\ y) \end{cases}$
- $\texttt{(cdr}\ l\texttt{)} \stackrel{\varepsilon}{\to} \begin{cases} \texttt{nil} & , l^\varepsilon = \texttt{nil} \\ y & , l^\varepsilon = (x\ .\ y) \end{cases}$

- `(null` l`)` $\stackrel{\varepsilon}{\to} \begin{cases} \texttt{t} & , l^\varepsilon = \texttt{nil} \\ \texttt{nil} & , \text{otherwise} \end{cases}$

- `(consp` l`)` $\stackrel{\varepsilon}{\to} \begin{cases} \texttt{t} & , l^\varepsilon = (x \ . \ y) \\ \texttt{nil} & , \text{otherwise} \end{cases}$

3.2.3 Structures

- `(struct` f arg_1 ... arg_n`)` $\stackrel{\varepsilon}{\to}$ `[`f^ε arg_1^ε ... arg_n^ε`]`, where f^ε is a symbol

- `(structp` s`)` $\stackrel{\varepsilon}{\to} \begin{cases} \texttt{t} & , s^\varepsilon = [f \ldots] \\ \texttt{nil} & , \text{otherwise} \end{cases}$

- `(arity [`f arg_1 ... arg_n`])` $\stackrel{\varepsilon}{\to}$ n

- `(functor [`f arg_1 ... arg_n`])` $\stackrel{\varepsilon}{\to}$ f

- `(elt [`f arg_0 ... arg_n`]` m`)` $\stackrel{\varepsilon}{\to}$ arg_m with $0 \leq m \leq n$

3.2.4 (Single and Re-) Assignments

- `(let ((`v_1 e_1`)` ... `(`v_n e_n`)) .` $body$`)` $\stackrel{\varepsilon}{\to}$ `(progn .` $body$`)`$^\varepsilon$
  where *body* is evaluated in the context of the local variables $v_i = e_i^\varepsilon$, $i \in \{1, \ldots, n\}$, created in *parallel*, i.e. first all e_i are evaluated, then the v_i are (single-) assigned (bound)

- `(let* ((`v_1 e_1`)` ... `(`v_n e_n`)) .` $body$`)` $\stackrel{\varepsilon}{\to}$ `(progn .` $body$`)`$^\varepsilon$
  where *body* is evaluated in the context of the local variables $v_i = e_i^\varepsilon$, $i \in \{1, \ldots, n\}$, created *sequentially*

- `(psetq` var_1 $expr_1$ ... var_n $expr_n$`)` $\stackrel{\varepsilon}{\to}$ `nil`
  with the *parallel* (re-)assignment of var_i to $expr_1^\varepsilon$, $i \in \{1, \ldots, n\}$, via side effect

- `(setq` var_1 $expr_1$ ... var_n $expr_n$`)` $\stackrel{\varepsilon}{\to}$ $expr_n^\varepsilon$
  with the *sequential* (re-)assignment of var_i to $expr_1^\varepsilon$, $i \in \{1, \ldots, n\}$, via side effect

3.2.5 Equality

- `(equal` x y`)` $\stackrel{\varepsilon}{\to} \begin{cases} \texttt{t} & , x^\varepsilon = y^\varepsilon \\ \texttt{nil} & , \text{otherwise} \end{cases}$

- `(eq` x y`)` $\stackrel{\varepsilon}{\rightarrow} \begin{cases} \texttt{t} & \text{, if the } \textit{internal} \text{ toplevel representation (pointer)} \\ & \quad \text{is equal (see section 5.4.4)} \\ \texttt{nil} & \text{, otherwise} \end{cases}$
- `(eql` x y`)` : same as `eq` (in COMMON LISP, `eq` is different from `eql` in that it is not guaranteed that `eq` works for numbers depending on their internal representation; in LL, `eq` and `eql` are identical since the abstract machine for LL, the LLAMA (see chapter 5), represents numbers just like all other constants)

3.2.6 Numerical and String Builtins

3.2.6.1 Numerical Builtins

- `(+` $arg_1 \ \ldots \ arg_n$`)` $\stackrel{\varepsilon}{\rightarrow} \sum_{i=1}^{n} arg_i^{\varepsilon},\ n \geq 0$
- `(-` $arg_0 \ \ldots \ arg_n$`)` $\stackrel{\varepsilon}{\rightarrow} \begin{cases} arg_0^{\varepsilon} - \sum_{i=1}^{n} arg_i^{\varepsilon} & , n \geq 1 \\ -arg_0^{\varepsilon} & , n = 0 \end{cases}$
- `(*` $arg_1 \ \ldots \ arg_n$`)` $\stackrel{\varepsilon}{\rightarrow} \prod_{i=1}^{n} arg_i^{\varepsilon},\ n \geq 0$
- `(/` $arg_0 \ \ldots \ arg_n$`)` $\stackrel{\varepsilon}{\rightarrow} \begin{cases} \frac{arg_0^{\varepsilon}}{\prod_{i=1}^{n} arg_i^{\varepsilon}} & , n \geq 1 \\ \frac{1}{arg_0^{\varepsilon}} & , n = 0 \end{cases}$
- `(1+` arg`)` $\stackrel{\varepsilon}{\rightarrow} arg^{\varepsilon} + 1$
- `(1-` arg`)` $\stackrel{\varepsilon}{\rightarrow} arg^{\varepsilon} - 1$
- `(=` $arg_0 \ \ldots \ arg_n$`)` $\stackrel{\varepsilon}{\rightarrow} \begin{cases} \texttt{t} & , \forall_{i=1}^{n}\ arg_0^{\varepsilon} = arg_i^{\varepsilon} \\ \texttt{nil} & , \text{otherwise} \end{cases}$
- `(/=` $arg_1 \ \ldots \ arg_n$`)` $\stackrel{\varepsilon}{\rightarrow} \begin{cases} \texttt{t} & , \forall_{i=1}^{n}\ \forall_{j=1}^{n}\ (i \neq j \Rightarrow arg_i^{\varepsilon} \neq arg_j^{\varepsilon}) \\ \texttt{nil} & , \text{otherwise} \end{cases}$
- `(<` $arg_1 \ \ldots \ arg_n$`)` $\stackrel{\varepsilon}{\rightarrow} \begin{cases} \texttt{t} & , \forall_{i=1}^{n-1}\ arg_i^{\varepsilon} < arg_{i+1}^{\varepsilon} \\ \texttt{nil} & , \text{otherwise} \end{cases}$
- `(>` $arg_1 \ \ldots \ arg_n$`)` $\stackrel{\varepsilon}{\rightarrow} \begin{cases} \texttt{t} & , \forall_{i=1}^{n-1}\ arg_i^{\varepsilon} > arg_{i+1}^{\varepsilon} \\ \texttt{nil} & , \text{otherwise} \end{cases}$
- `(<=` $arg_1 \ \ldots \ arg_n$`)` $\stackrel{\varepsilon}{\rightarrow} \begin{cases} \texttt{t} & , \forall_{i=1}^{n-1}\ arg_i^{\varepsilon} \leq arg_{i+1}^{\varepsilon} \\ \texttt{nil} & , \text{otherwise} \end{cases}$
- `(>=` $arg_1 \ \ldots \ arg_n$`)` $\stackrel{\varepsilon}{\rightarrow} \begin{cases} \texttt{t} & , \forall_{i=1}^{n-1}\ arg_i^{\varepsilon} \geq arg_{i+1}^{\varepsilon} \\ \texttt{nil} & , \text{otherwise} \end{cases}$

3.2.6.2 String Builtins

- `(string<` arg_1 arg_2`)` $\stackrel{\varepsilon}{\rightarrow} \begin{cases} \texttt{t} & , arg_1^\varepsilon \text{ is lexicographically less than } arg_2^\varepsilon \\ \texttt{nil} & , \text{otherwise} \end{cases}$

- `(string>` arg_1 arg_2`)` $\stackrel{\varepsilon}{\rightarrow} \begin{cases} \texttt{t} & , arg_1^\varepsilon \text{ is lexicographically greater than } arg_2^\varepsilon \\ \texttt{nil} & , \text{otherwise} \end{cases}$

3.2.7 I/O Builtins

- `(read)` $\stackrel{\varepsilon}{\rightarrow}$ *input*, where *input* is any LL term entered by the user

- `(print` *expr*`)` $\stackrel{\varepsilon}{\rightarrow}$ $expr^\varepsilon$; $expr^\varepsilon$ is printed as side effect

3.2.8 Higher-order Builtins

- `(funcall` *function* $expr_1$ ... $expr_n$`)`:
  function application of $function^\varepsilon$ to `(`$expr_1^\varepsilon$ ... $expr_n^\varepsilon$`)`

- `(apply` *function* $expr_1$ ... $expr_{n-1}$ $expr_n$`)`:
  function application of $function^\varepsilon$ to `(`$expr_1^\varepsilon$ ... $expr_{n-1}^\varepsilon$ `.` $expr_n^\varepsilon$`)`;
  $expr_n^\varepsilon$ must be a list

- `(function` *function*`)` or `#'`*function*: evaluates to the functional object corresponding to *function* in the current environment; *function* may be a symbol or a lambda expression `(lambda (`v_1 ... v_n`)` `.` *body*`)`

- `(eval` *expr*`)`: *expr* is evaluated twice; the first evaluation takes place in the current environment, the second in the global environment

3.2.9 Sequential Evaluation

- `(progn` $expr_1$ ... $expr_n$`)` $\stackrel{\varepsilon}{\rightarrow}$ $expr_n^\varepsilon$; $expr_1$ ... $expr_{n-1}$ are evaluated for their side effects

3.2.10 Conditional Evaluation

- `(if` *test* $expr_1$ $expr_2$`)` $\stackrel{\varepsilon}{\rightarrow} \begin{cases} expr_1^\varepsilon & , test^\varepsilon \neq \texttt{nil} \\ expr_2^\varepsilon & , \text{otherwise} \end{cases}$
  `(if` *test* *expr*`)`$^\varepsilon$ = `(if` *test* *expr* `nil)`$^\varepsilon$

- `(cond (`t_1 `.` e_1`)` ... `(`t_n `.` e_n`))` $\stackrel{\varepsilon}{\rightarrow}$ `(progn .` e_i`)`$^\varepsilon$ with $t_i^\varepsilon \neq \texttt{nil} \wedge \forall_{j=1}^{i-1}\, t_j^\varepsilon = \texttt{nil}$

- (and $expr_1$ ... $expr_n$) $\stackrel{\varepsilon}{\rightarrow} \begin{cases} \texttt{t} & , n = 0 \\ expr_n^\varepsilon & , n > 0 \land \forall_{i=1}^n \, expr_i^\varepsilon \neq \texttt{nil} \\ \texttt{nil} & , \text{otherwise} \end{cases}$

- (or $expr_1$ ... $expr_n$) $\stackrel{\varepsilon}{\rightarrow} \begin{cases} expr_i^\varepsilon & , \exists_{i \in \{1,\ldots,n\}} \, expr_i^\varepsilon \neq \texttt{nil} \\ & \quad \land \forall_{j=1}^{i-1} \, expr_j^\varepsilon = \texttt{nil} \\ \texttt{nil} & , \text{otherwise} \end{cases}$

3.2.11 Non-local Exits and Loops

Non-local exits (`catch` and `throw`) and loops (`loop` and `do`) have been provided for two reasons:

1. to make the integration of existing COMMON LISP programs into REL programs easier
2. to allow tail-recursive deterministic REL relations and functions to be translated into LL loops in order to speed up execution

For a detailed explanation of `catch`, `throw`, `loop`, and `do`, see [Steele Jr., 1984]. Here a short description:

- (catch *tag* . *body*) $\stackrel{\varepsilon}{\rightarrow} \begin{cases} (\texttt{progn} \ . \ body)^\varepsilon & , \text{if no } \texttt{throw} \text{ statement was evaluated in } body \\ expr^\varepsilon & , \text{if } (\texttt{throw} \ tag \ expr) \text{ was evaluated in } body \end{cases}$

- (throw *tag* *expr*): leave the next `catch` body tagged with tag^ε, returning $expr^\varepsilon$

- (loop . *body*): repeat executing *body*; returns only if (throw nil *expr*) is executed (this is slightly different from COMMON LISP `loop`)

- (return *expr*): same as (throw nil *expr*)

- (do *init* *exit* . *body*)
  with $init = ((var_1^I \ expr_{1_a}^I \ expr_{1_b}^I)\ldots(var_n^I \ expr_{n_a}^I \ expr_{n_b}^I))$,
  $exit = (test \ expr_1^E \ldots expr_m^E)$

  The execution of a `do` loop takes place as follows:

  1. initialize var_i^I with ${expr_{i_a}^I}^\varepsilon$ via `let`
  2. if $test^\varepsilon \neq$ `nil`, return $(\texttt{progn} \ expr_1^E \ldots expr_m^E)^\varepsilon$
  3. if $test^\varepsilon =$ `nil`, execute *body* and re-initialize var_i^I with ${expr_{i_b}^I}^\varepsilon$ via `psetq`; go to 2.

3.2.12 Prelude

- $\texttt{c}\diamond\texttt{r}$ with $\diamond \in \{aa, ad, da, dd\}$:
  $(\texttt{c}\diamond\texttt{r}\ l)^\varepsilon = (\texttt{c}\diamond_1\texttt{r}\ (\texttt{c}\diamond_2\texttt{r}\ l))^\varepsilon$ for $\diamond = \diamond_1\diamond_2$

- $(\texttt{mapcar}\ \mathit{function}\ \mathit{list}) \xrightarrow{\varepsilon} (y_1\ \ldots\ y_n)$
  $$\text{with}\quad \mathit{list}^\varepsilon = (x_1\ \ldots\ x_n) \wedge \forall_{i=1}^{n}\ (\texttt{funcall}\ \mathit{function}\ \texttt{'}x_i)^\varepsilon = y_i$$

- $(\texttt{append}\ \mathit{list}_1\ \mathit{list}_2) \xrightarrow{\varepsilon} (x_1\ \ldots\ x_n\ y_1\ \ldots\ y_m)$
  $$\text{with}\quad \begin{aligned} \mathit{list}_1^\varepsilon &= (x_1\ \ldots\ x_n) \wedge \\ \mathit{list}_2^\varepsilon &= (y_1\ \ldots\ y_m) \end{aligned}$$

- $(\texttt{reverse}\ \mathit{list}) \xrightarrow{\varepsilon} (x_n\ \ldots\ x_1)$ with $\mathit{list}^\varepsilon = (x_1\ \ldots\ x_n)$

- $$(\texttt{member}\ x\ \mathit{list}) \xrightarrow{\varepsilon} \begin{cases} (x\ y_p \ldots\ y_n) & ,\ \mathit{list}^\varepsilon = (y_1\ \ldots\ y_n) \\ & \quad \wedge\ \exists_{p\in\{1,\ldots,n\}}\ x^\varepsilon = y_p \\ & \quad \wedge\ \forall_{j=1}^{p-1}\ x^\varepsilon \neq y_j \\ \texttt{nil} & ,\ \text{otherwise} \end{cases}$$

- $$(\texttt{assoc}\ x\ \mathit{alist}) \xrightarrow{\varepsilon} \begin{cases} (k_p\ .\ v_p) & ,\ \mathit{alist}^\varepsilon = ((k_1\ .\ v_1)\ \ldots\ (k_n\ .\ v_n)) \\ & \quad \wedge\ \exists_{p\in\{1,\ldots,n\}}\ x^\varepsilon = k_p \\ & \quad \wedge\ \forall_{j=1}^{p-1}\ x^\varepsilon \neq k_j \\ \texttt{nil} & ,\ \text{otherwise} \end{cases}$$

- $(\texttt{sort}\ \mathit{list}\ \mathit{pred}) \xrightarrow{\varepsilon} (x_{i_1}\ \ldots\ x_{i_n})$
  $$\text{with}\quad \mathit{list}^\varepsilon = (x_1\ \ldots\ x_n) \wedge \{i_1, \ldots, i_n\} = \{1, \ldots, n\} \wedge \forall_{j=1}^{n-1}\ (\texttt{funcall}\ \mathit{pred}\ \texttt{'}x_{i_j}\ \texttt{'}x_{i_{j+1}})^\varepsilon \neq \texttt{nil}$$

3.3 User-definable Functions

In LL, functions can only be defined via `defun` in the following form:

(`defun` *function* (*var*$_1$ ... *var*$_n$) . *body*)

Optional, keyword, and rest parameters are not directly supported. However, in contrast to COMMON LISP, a function may have — just like a REL procedure — more than one arity. This allows REL procedures to be transformed into LL functions more easily because renaming is not necessary.

Example 3.1 (LL function with more than one arity)

`(defun incadd (x) (1+ x))`	`(incadd 2)` $\xrightarrow{\varepsilon}$ `3`
`(defun incadd (x y) (+ x y))`	`(incadd 1 2)` $\xrightarrow{\varepsilon}$ `3`

△

This also allows optional and keyword parameters to be simulated. For instance, the COMMON LISP function definition

```
(defun f (x y &optional (z 0)) ... )
```

can be expressed in LL as:

```
(defun f (x y z) ... )
(defun f (x y) (f x y 0))
```

Similarly, 'finite rests' (e.g. a fixed number, n, of arities), can be expressed:

```
(defun f () ...)
(defun f (x1) ...)
...
(defun f (x1 ... xn) ...)
```

Chapter 4

Integrating Relations and LL

In this chapter, the three facets of the integration of relational languages and LL are described:

1. their integration by loose coupling on the abstract machine and source language level,

2. the transformation of deterministic predicates into LL functions, and

3. the implementation of extra-logicals with the help of LL.

In section 4.1, the loose coupling of REL and LL is described, where the emphasis is on the accessibility of LL functions from predicates (since the other direction is not part of our prototypical implementation).

In section 4.2, the detection and transformation of deterministic predicates is portrayed. For this, first a theoretical foundation of determinism in relational languages is developed (using SLD resolution trees), and then algorithms for the detection and transformation of deterministic predicates, using several intermediate representation languages, are given.

In section 4.3, the implementation of extra-logicals in relational languages via LL functions is described by giving a simple example (global variables and, building upon them, `bagof`).

4.1 Loose Coupling of Relations and LL

Relational programs and LL programs principally access each other in two different ways:

1. LL functions are called from relations

2. relations are called from LL functions

4.1.1 Calling LL Functions from Relations

On the *source language level*, the access of LL functions from relations is performed by a generalized `is` builtin: in addition to arithmetical builtins, functions defined in LL may be used on the right-hand side of `is` premises. Furthermore, LL (test) predicates can be used as guards in relational clauses. An LL predicate is a boolean-valued function for which a returned `nil` is interpreted as *false* (failure) and any other value as *true*.

The following example illustrates this:

Example 4.1

```
(defun append (l1 l2)
  (if (null l1)
      l2
      (cons (car l1) (append (cdr l1) l2))))

(defun member (x l)
  (if l (or (equal x (car l)) (member x (cdr l)))))

p(X,L1,L2,R) :-
  member(X,L1),
  R is append(L1,L2).
p(X,L1,L2,R) :-
  member(X,L2),
  R is append(L2,L1).
```

The following queries are now possible:

```
rfe-p> p(3, [1,2,3], [4,5,6], R)
true
R = [1, 2, 3, 4, 5, 6]
rfe-p> more
unknown
rfe-p> p(3, [4,5,6], [1,2,3], R)
true
R = [1, 2, 3, 4, 5, 6]
rfe-p> more
unknown
rfe-p> p(3, [1,2,3], [3,4,5], R)
true
R = [1, 2, 3, 3, 4, 5]
rfe-p> more
true
R = [3, 4, 5, 1, 2, 3]
```

```
rfe-p> more
unknown
rfe-p> p(3, [1,2], [4,5])
unknown
```

△

On the *abstract machine level*, the accessibility of LL functions (which are compiled into LLAMA code) from predicates (which are compiled into WAM code) is achieved by extending the WAM with two new instructions:

1. `ll` is used to execute an ordinary LL function
2. `llp` is used to execute an LL predicate

Both `ll` and `llp`, when executing an n-ary LL function/predicate, push the first n WAM argument registers on the stack (this is necessary since the abstract machine behind LL, the LLAMA, is a stack machine; see chapter 5). Note that by simply copying the contents of WAM registers only the toplevel representation (pointers) of complex data structures (lists and structures) are copied, not the data structures themselves.

After executing the LLAMA code, the result, which is the topmost element on the stack, is stored in the first WAM argument register `X1` which is then unified with the left-hand side of the `is` call. In case of an LL predicate, a *fail* is generated if the return value is `nil`.

4.1.2 Calling Relations from LL Functions

Since LL does not support backtracking, relations can only be accessed in the following two ways from LL functions:

1. the relation returns only the first solution (`once` mode)
2. the relation returns a list of all solutions (`bagof` mode)

The extensions of the LLAMA are similar to those needed for calling LL functions from relations.

4.2 Determinism Transformation

Real-world applications of relational languages tend to contain a very large portion of deterministic procedures for several reasons:

- potentially invertable predicates are often not meaningfully inverted, e.g. `reverse`, or used only *in one direction*, e.g. `append`
- most *mathematical computations* (as they occur in applications dealing with economics, statistics, technical fields, etc.) are deterministic
- of course, *user interaction*, *file I/O*, and other communications with the underlying operating system are deterministic[1]

For most of these deterministic predicates it is important to be highly efficient. Besides attempts at efficient compilation of deterministic PROLOG itself [Van Roy, 1994], this can be achieved by either implementing deterministic predicates in a loosely coupled imperative or functional programming language (as described in section 4.1) or by specifying them in a relational language with subsequent transformation into an efficient (e.g. imperative or functional) language by an intelligent compiler.

Loose coupling has often the disadvantage of not maintaining full *declarativeness*: the application has to be split into parts specified in different paradigms of different levels. A much more declarative approach is to implement the complete program in a relational language and then leave it to the compiler to achieve the desired efficiency.

In the following subsections, a set of transformations and intermediate languages is described which are used to detect and compile deterministic relations into LL.

4.2.1 Determinism in Relational Languages

Before describing the algorithms for determinism transformation, *determinism in relational languages*[2] is defined. The definition presented in this work was designed to cover as many deterministic predicates as possible since in the context of a transformation into LL, the gain in efficiency grows directly with the number of transformed deterministic predicates.

In this work, determinism is defined via *SLD_{PROLOG} resolution trees*:

Definition 4.1 ($\mathbf{SLD}_{PROLOG}$ Resolution Tree)
An SLD_{PROLOG} resolution tree *for a program P and an atomic*[3] *goal g is the*

[1] For reasons of simplicity, relational programs with side effects will not be covered in this work.

[2] In this work, only relational languages with PROLOG's computation and search rule are considered.

[3] Only atomic goals $g(\ldots)$ are considered; complex, conjunctive, goals $g_1(\ldots), \ldots, g_n(\ldots)$ can be transformed into atomic goals by adding $g(v_1, \ldots, v_m) :- g_1(\ldots), \ldots, g_n(\ldots)$ to P and using $g(v_1, \ldots, v_m)$ as the new goal (the v_i, $i \in \{1, \ldots, m\}$, are the variables of the original goal $g_1(\ldots), \ldots, g_n(\ldots)$).

partial SLD resolution tree [Lloyd, 1987] for g in P created according to the PROLOG[4] *refutation procedure, i.e. with PROLOG's computation rule (which always selects the leftmost goal in a goal sequence) and PROLOG's depth-first search rule (searching clauses from top to bottom).*

Nodes are either labeled with a goal sequence $\langle g_1, \ldots, g_n \rangle$ *or the empty goal* $\langle\rangle$ *or* $\Box$ *to indicate success.* $\triangle$

Definition 4.2 (Multiset of Nodes $\mathcal{N}^P(g)$)
For a program P and an atomic goal g, $\mathcal{N}^P(g)$ *is the (finite or infinite)* multiset of all nodes *in the* SLD_{PROLOG} *resolution tree for P and g.* $\triangle$

Definition 4.3 (Activated Subgoals $\mathcal{S}^P(g)$)
For a program P and an atomic goal g, the set of all activated subgoals of g w.r.t. *P is defined as*

$$\mathcal{S}^P(g) \ := \ \{g' \ \mid \ \exists n \in \mathcal{N}^P(g) : n = \langle g', \ldots \rangle\}$$

For a program P and a set of goals G, the set of all activated subgoals of G w.r.t. *P is defined as*

$$\mathcal{S}^P(G) \ := \ \bigcup_{g \in G} \mathcal{S}^P(g)$$

$\triangle$

As this definition states, all goals appearing as *first* elements in the nodes of an SLD_{PROLOG} resolution tree are activated subgoals. This is due to PROLOG's computation rule which always selects the leftmost goal in a goal sequence.

Definition 4.4 (Predicate Symbol $\pi(g)$)
For an atomic goal g, the predicate symbol $\pi(g)$ *is defined as*

$$\pi(g) = p \iff (g = p(t_1, \ldots, t_n) \wedge p \neq \texttt{once}) \ \vee \ g = \texttt{once}(p(t_1, \ldots, t_n))$$
(where the t_i, $i \in \{1, \ldots, n\}$, *are any PROLOG terms)*

$\triangle$

Now the fundamental definition of determinism in PROLOG can be given:

Definition 4.5 (Deterministic PROLOG Predicate)
Let P be a PROLOG program and Q a set of atomic queries for P. A predicate p in P is called deterministic w.r.t. P and Q *iff*[5]

$$\begin{array}{llll} \forall q \in \mathcal{S}^P(Q): & (\pi(q) = p & \Rightarrow & (|\mathcal{N}^P(q)| < \infty \wedge |\mathcal{N}^P(q)|_{\Box} \leq 1) \\ & & & \vee(|\mathcal{N}^P(q)| = \infty \wedge \Box \notin \mathcal{N}^P(q)) \quad) \end{array}$$

$\triangle$

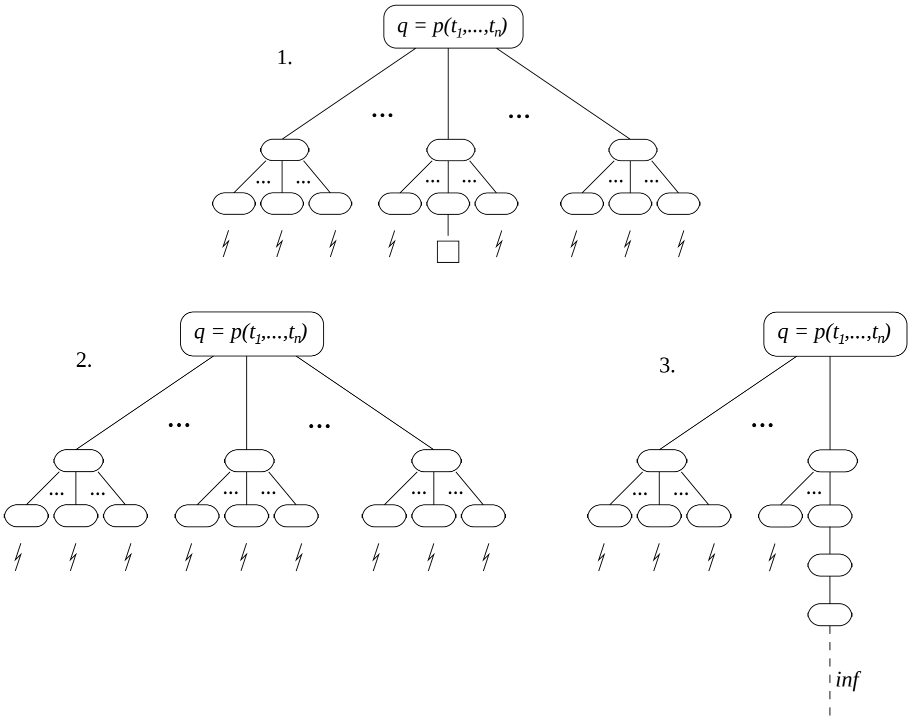
[4] In this work, only PROLOG with cut and `once` is considered; negation and other extra-logicals are not taken into account.

[5] $|MS|_x$ is the number of occurrences of x in the multiset *MS*.

This definition specifies a predicate p to be deterministic if for all possible queries all activated subgoals q with p as predicate either

1. have exactly one solution,
2. fail, or
3. enter a non-terminating branch (before computing a first solution)[6]

when called separately.



There are other possibilities to define determinism in PROLOG[7]. In the following, examples motivating our definition and modifications that make it tractable and more powerful are given.

In definition 4.5, a predicate is deterministic w.r.t. to a given program P and a given set of queries Q. A definition of determinism in PROLOG that does not consider an entire program and a set of queries but only the clauses of a single predicate will not be fruitful: only very few PROLOG predicates are deterministic as such, and — as already mentioned — as many predicates as possible should be covered in order to gain efficiency by transforming them into LL.

---

[6]Thus non-determinism also covers cases where a predicate is first successful and then enters a non-terminating branch.

[7]Here, determinism is defined *dynamically* via proof trees. A more common approach is to define determinism *statically*, e.g. via non-overlapping clause heads.

Example 4.2 Let us consider the following simple program, only containing the predicate `p`:

```
p(1).
p(2).
p(3).
```

While `p` is of course deterministic for ground queries, it is not deterministic for general queries: the non-ground query `p(X)` has three solutions.

If and only if we restrict the set of possible queries to `p(`*nvterm*`)` with *nvterm* being any PROLOG term except a (free) variable, then `p` is deterministic: the SLD_{PROLOG} derivation trees for all queries either directly end with a fail or with the empty goal. For "non-variable queries", `p` could thus be replaced by the following (correctness-preserving) LL function, where `nil` resulting from the `or` for $\mathtt{x} \notin \{1, 2, 3\}$ is interpreted as fail:

```
(defun p (x) (or (equal x 1) (equal x 2) (equal x 3)))
```
$\triangle$

Example 4.2 shows that it is important to consider all queries for a given program. Otherwise, predicates that are always used deterministically are not detected and thus compiled into inefficient code.

This definition complies with the concept of *partial evaluation* (or *partial deduction* in the field of logic programming). As pointed out in [Nilsson, 1993], partial evaluation is, at least from a principal point of view, a simple form of program transformation by which a program, given some *partial input data*, can be specialized to solve a particular problem more efficiently than the more general program. The origin of the field is due to Kleene [Kleene, 1952], but the application to programming languages is primarily due to Futamara [Futamura, 1971], Haraldsson [Haraldsson, 1977], Ershov [Ershov, 1980], and Jones [Jones *et al.*, 1989].

As the following example shows, non-trivial deterministic predicates can be detected without partial evaluation.

Example 4.3 (`once` Determinism)
For a given program P and a query set Q, a predicate p is `once` deterministic *iff*

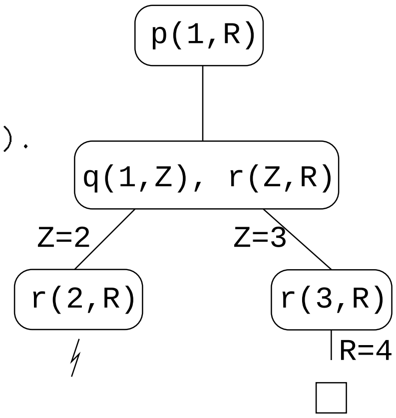
$$\forall q \in \mathcal{S}^P(Q) : (\pi(q) = p \Rightarrow q = \mathtt{once}(p(\ldots)))$$

Once determinism directly fulfills the definition of determinism. In the special case that a predicate is always embedded into an explicit `once` in all queries of Q and in all premises in P, it satisfies the definition of `once` determinism. This form of determinism can be detected by simple syntactical inspection of P and Q; partial evaluation is not needed. $\triangle$

The next example shows that it is not sufficient for a predicate to have at most one solution for all activated subgoals to be *easily* replaced by an efficient deterministic procedure.

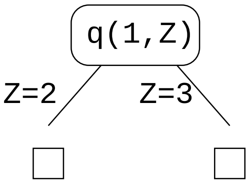
Example 4.4

```
p(X,Y) :- q(X,Z), r(Z,Y).
q(1,2).
q(1,3).
r(3,4).
r(4,5).
```



For $Q = \{$`p(1,R)`$\}$, `p` is deterministic according to definition 4.5. $\triangle$

The problem with this program is that although `p` is deterministic w.r.t. P and Q, `q`, which is called in `p`, is non-deterministic w.r.t. P and $Q' = \{$`q(1,Z)`$\}$ $\subset \mathcal{S}^P($`p(1,R)`$)$:



In order to be able to replace each deterministic predicate by a simple and directly obtainable procedure or function specified in a deterministic imperative or functional language, all predicates called in the execution of deterministic predicates must themselves be deterministic: *internal backtracking* in the execution of a deterministic predicate is not allowed.

This leads to the following definition of *deep determinism*:

Definition 4.6 (Deeply Deterministic PROLOG Predicate)
Let P be a PROLOG program and Q a set of atomic queries for P. A predicate p in P is called deeply deterministic w.r.t. P and Q *iff*

$$\begin{aligned} & p \text{ is deterministic w.r.t. } P \text{ and } Q \\ \wedge\ & \forall q \in \mathcal{S}^P(Q) : (\pi(q) = p \Rightarrow \forall r \in \mathcal{S}^P(q) \backslash \{q\} : \pi(r) \text{ is deeply deterministic} \\ & \text{w.r.t. } P \text{ and } Q' = \{r\}) \end{aligned}$$

$\triangle$

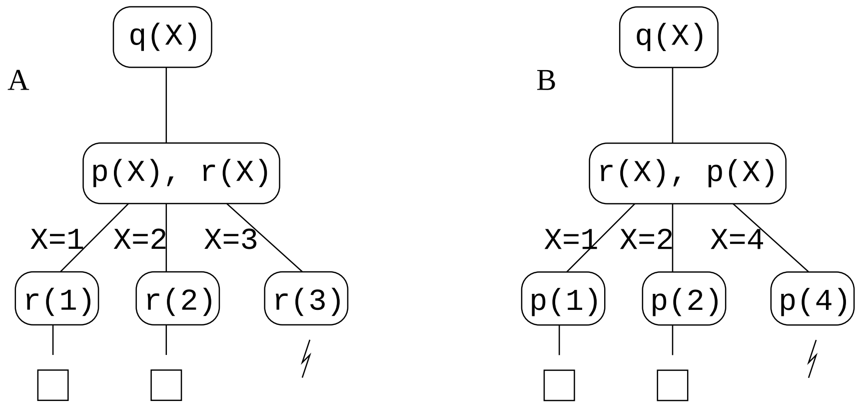
An algorithm detecting deeply deterministic predicates (for a special case of query sets) is described in section 4.2.3.

The following example motivates a further modification of the concept of deterministic predicates:

Example 4.5 Let us consider two *logically equivalent* programs A and B:

A		B	
`q(X) :- p(X), r(X).`		`q(X) :- r(X), p(X).`	
`p(1).`	`r(1).`	`p(1).`	`r(1).`
`p(2).`	`r(2).`	`p(2).`	`r(2).`
`p(3).`	`r(4).`	`p(3).`	`r(4).`

If the only query is `q(X)`, then the following SLD_{PROLOG} derivation trees result:



In program A, only `r` is (deeply) deterministic, while in B only `p` is (deeply) deterministic:

1. In A, the set of all activated subgoals $\mathcal{S}$(`q(X)`) is {`q(X)`, `p(X)`, `r(1)`, `r(2)`, `r(3)`}. Of these, `q(X)` has two and `p(X)` has three solutions, whereas `r(1)` and `r(2)` have one and `r(3)` has no solutions, which makes `r` the only (deeply) deterministic predicate.

2. In B, `p` is derived to be the only (deeply) deterministic predicate in analogy to 1.

△

This example shows that due to PROLOG's computation rule the set of deterministic predicates in a program can change if the program is transformed into a logically equivalent program. This motivates a slight modification of definitions

4.5 and 4.6: in order to obtain more (or more interesting, e.g. more often used) deterministic predicates, reordering of premises should be allowed.

Reordering of premises is potentially dangerous in PROLOG for several reasons:

1. the termination behaviour may change
2. solutions may be computed in a different order
3. side effects may be executed in the wrong order

The transformation algorithms described in this work will only reorder premises in respective benevolent cases:

1. by reordering premises, in PROLOG non-terminating relations are sometimes transformed into terminating ones, but never the other way around

2. only premises of deeply deterministic predicates are reordered, which does not change the order of the computed solutions because deeply deterministic predicates have at most one solution

3. the transformations are not applied to programs with side effects

4.2.2 Restricting the Query Set

As was shown in the previous subsection, determinism analysis requires to consider not only the clauses of a single predicate but also the whole program and *a set of queries*. This gives rise to two problems:

1. usually the set of all queries is not known for a program
2. the set of all queries is infinite in most cases

These problems can be solved in the following ways:

1. The set of all possible queries is not supplied by the user but determined by global analysis. In this case, only very few restrictions of the query set can be found: predicates with numerical or other type restricted builtins are used to determine the types of some variables which are then propagated through the whole program by abstract interpretation. The (usually infinite) set of possible queries is then represented by a finite set of type restrictions.

2. The user declares types and/or modes for some or all predicates. Modes specify whether an argument is always *free*, *bound*, or *ground* when a predicate is called. These types and modes are then propagated by abstract interpretation to obtain a finite set of type and mode restrictions for as many predicates as possible.

In this work, a simplified version of the second solution was chosen: since PROLOG is (usually) untyped, only mode declarations are considered. A mode analyzer, e.g. the one described in [Krause, 1991], can then be used to refine the set of mode descriptions, thus further restricting the query set.

In the following, a set of mode descriptions, either only declared by the user or refined with a mode analyzer, is assumed to be present. Only two modes are used: *ground*, which is denoted by $\mathcal{G}$, and *any* (i.e. unrestricted), which is denoted by $\mathcal{X}$. For a predicate p, the mode declaration is of the form $p(\mu_1, \ldots, \mu_n)$ with $\mu_i \in \{\mathcal{G}, \mathcal{X}\},\ i \in \{1, \ldots, n\}$.

4.2.3 Transforming Relational Languages into LL

In this subsection, the determinism analysis and the transformation steps for deeply deterministic predicates are described.

For reasons of simplicity, only the following two kinds of deeply deterministic predicates are supported:

1. *functional* predicates: predicates which for a given set of ground input arguments compute a ground set of output arguments, i.e. they always compute exactly *one* solution and never fail (total functions)

2. *test* predicates: predicates which when called with all arguments ground either fail or succeed exactly once; these test predicates are used as *guards*

For a program P, in which the predicates $\mathcal{P}$ are defined, and a set of mode declarations M, deeply deterministic functional or test predicates are detected by the following algorithm:

1. $\mathcal{C} \leftarrow \{p \in \mathcal{P} \mid \exists p(\mu_1, \ldots, \mu_n) \in M\ \exists i \in \{1, \ldots, n\} : \mu_i = \mathcal{G}\}$:
   construct a set of candidates $\mathcal{C}$ consisting of all predicates with at least one ground argument (if all arguments are ground, the predicate is assumed to be a test predicate, otherwise it is assumed to be a functional predicate)

2. while $\exists p \in \mathcal{C} : (p(\ldots) :- \ldots, q, \ldots .) \in P\ \wedge\ \pi(q) \notin \mathcal{C}$ do $\mathcal{C} \leftarrow \mathcal{C} \backslash \{p\}$:
   repeat deleting all predicates in $\mathcal{C}$ if in their definition a predicate in $\mathcal{P} \backslash \mathcal{C}$ is used until no such predicate in $\mathcal{C}$ exists

3. try to transform all predicates in $\mathcal{C}$ into LL (with the algorithms described in the following subsections); remove all predicates from $\mathcal{C}$ which could not be thus transformed

4. *same as* 2.

In the transformation steps, several intermediate representations are used:

REL	input
RELFUN	functions
$(\mathcal{E}, v)$	expression environment (plus return value)
Λ	LL clause specification in RELFUN syntax
LL	output

The following REL example will be used to demonstrate the various transformations and intermediate representations. It consists of two deeply deterministic functional predicates:

Example 4.6

```
1a) fac(0,1) :- !.
 b) fac(X,Y) :-
      X1 is -(X,1),
      fac(X1, FX1),
      Y is *(X, FX1).

2a) f(s[X,Y], u[s[X,Y],s[A,B]]) :-
      A is -(B,A),
      s[A,B] is s[X,Y],
      fac(X,A),
      fac(Y,B) !.
 b) f(X, [X,X]).
```

The query set is represented by the following mode declarations: `fac(`$\mathcal{G}$`,` $\mathcal{X}$`)` and `f(`$\mathcal{G}$`,` $\mathcal{X}$`)`. △

This example was chosen for the following reasons:

1. `fac` (the factorial function) is used to show how the algorithms work for a *simple* case.
2. `f` is a rather artificial function using `fac`. It is used to show how the algorithms handle *non-standard* cases:
   - the definition of `f` cannot be evaluated by REL (the arithmetical builtin `-` is called with free variables),
   - the expression `A is -(B,A)` is cyclic,
   - the unification in `s[A,B] is s[X,Y]` has to be handled at compile time (*static unification*),
   - lists and structures are transformed into LL constructors, selectors, and test predicates, and
   - subexpressions in the output value can be shared with the input value.

4.2.3.1 REL → RELFUN Functions

In this first step, all deeply deterministic *functional* predicates are transformed into functional RELFUN specifications. For our example, this results in:

```
1+) fac(X1, X2) :- X2 is fac/2-1(X1).

1a) fac/2-1(0) :- !& 1.

1b) fac/2-1(X) :-
      X1 is -(X, 1),
      FX1 is fac/2-1(X1),
      Y is *(X, FX1)
      & Y.

2+) f(X1, X2) :- X2 is f/2-1(X1).

2a) f/2-1(s[X, Y]) :-
      A is -(B, A),
      s[A, B] is s[X, Y],
      A is fac/2-1(X),
      B is fac/2-1(Y) !
      & u[s[X, Y], s[A, B]].

2b) f/2-1(X) :-& [X, X].
```

The rules `1+` and `2+` are needed in order to allow to call the binary `fac` and `f` predicates from the toplevel. The original predicates are transformed into unary function definitions, renaming them by adding the suffix `/2-1`.

In general, this suffix has the form `/`m`-`n where m is the original arity and n is the number of output arguments (specified by $\mathcal{X}$ in the mode declaration). For $n > 1$, the output values are embedded in a `values` structure, e.g.

`f(X,Y,A,B) :- A is +(X,Y), B is *(X,Y).`

with the modes `f(`$\mathcal{G}$`, `$\mathcal{G}$`, `$\mathcal{X}$`, `$\mathcal{X}$`)` is transformed into

`f(X,Y,A,B) :- values[A,B] is f/4-2(X,Y).`

`f/4-2(X,Y) :- A is +(X,Y), B is *(X,Y) & values[A,B].`

This also works for non-adjacent output arguments: `g(X,A,Y,B) :- ...` with modes `g(`$\mathcal{G}$`,`$\mathcal{X}$`,`$\mathcal{G}$`,`$\mathcal{X}$`)` is transformed into `g/4-2(X,Y) :- ... & values[A,B]`.

Such value structures can be viewed as representing multiple-valued functions as described in [Stein and Sintek, 1991].

4.2.3.2 RELFUN → $(\mathcal{E}, v)$

The next transformation step takes all deterministic clauses totally apart: an internal representation, called $\mathcal{E}$, is created, in which variables are associated with expressions ($\mathcal{E}$ stands for *environment* because this data structure strongly resembles the sort of environments used for unification in PROLOG interpreters). The return value is represented by v.

This structure is needed for several reasons:

- all arguments of subqueries have to be ground; this often requires reordering of premises
- unification has to be done at compile time (*static unification*), resulting in assignments and equality tests
- in LL, the arguments of a function have to be distinct variables and cannot be arbitrary terms as in PROLOG
- common subexpressions are (statically) unified, e.g. if `X is f(Z)` and `Y is f(Z)` occur in a clause, then `X` is unified with `Y`, e.g. by replacing `Y` by `X` everywhere in the clause (this is allowed because predicates with side effects and non-deterministic predicates are not considered)

For our example, $(\mathcal{E}, v)$ looks like this:

1a	$\mathcal{E}$	`Arg#1`	`0`
	v	`1`	
1b	$\mathcal{E}$	`X1`	`-(Arg#1, 1)`
		`FX1`	`fac/2-1(X1)`
		`Y`	`*(Arg#1, FX1)`
	v	`Y`	
2a	$\mathcal{E}$	`V1`	`struct(u, Arg#1, Arg#1)`
		`V2`	`structp(Arg#1)`
		`X`	`elt(Arg#1, 0), fac/2-1(X), -(Y, X)`
		`Y`	`elt(Arg#1, 1), fac/2-1(Y)`
		`V3`	s, `functor(Arg#1)`
		`V4`	2, `arity(Arg#1)`
	v	`V1`	
2b	$\mathcal{E}$	`V1`	`cons(Arg#1, V2)`
		`V2`	`cons(Arg#1, nil)`
	v	`V1`	

Note that in $(\mathcal{E}, v)$ a variable need not be associated with a unique expression, e.g. in 2a, `X` is associated with three and `Y` with two expressions. As will be shown in the following subsection, exactly one of the expressions associated with a variable is used to *bind* the variable (single assignment); the remaining expressions are only *compared* with the variable (if one of the expressions a variable is associated with is a constant, then the variable is not needed: it is sufficient to compare all remaining expressions with the constant, e.g. `V3` and `V4`).

$(\mathcal{E}, v)$ was created by (local) abstract interpretation of each clause using the following techniques:

- *normalizing heads*: all head terms are moved into the body, leaving `Arg#`i as the only head terms
- *normalizing lists and structures*: all lists and structures are transformed into the LL constructors, selectors, and test predicates (`cons`, `car`, `struct`, etc.)
- *static unification*: all unification is translated into environment entries in $\mathcal{E}$ just as a concrete PROLOG interpreter would do: in clause `2a`, `s[A, B] is s[X, Y]` has been used to unify `A` with `X` and `B` with `Y`
- *unification of common subexpressions*: static unification is not only used for explicit unification (`is`) but also for common subexpressions: the output value of clause `2a`, `V1`, is built using the original input argument `Arg#1` without re-creating `s[X,Y]` and `s[A,B]`: `V1 = struct(u, Arg#1, Arg#1)`

4.2.3.3 $(\mathcal{E}, v) \rightarrow \Lambda$

The next transformation step transforms clauses represented as $(\mathcal{E}, v)$ into Λ clauses (only information contained in $(\mathcal{E}, v)$ is needed). Λ clauses are RELFUN clauses with the following features:

- structures and lists are represented by LL constructors, selectors, and test predicates,
- the premises have already been ordered such that all arguments are ground and test predicates come before selectors,
- unification (i.e., each entry in $\mathcal{E}$) has been transformed into assignments (via `is`) and equality tests (via `equal`),
- expressions have been inserted as arguments for other expressions when possible, and
- all cuts are removed (this is allowed since Λ clauses are only used as an *intermediate* representation).

For our example, the following Λ clauses created:

```
fac/2-1(Arg#1) :- equal(0, Arg#1) & 1.
fac/2-1(Arg#1) :-& *(Arg#1, fac/2-1(-(Arg#1, 1))).
```

```
f/2-1(Arg#1) :-
        structp(Arg#1),
        equal(s, functor(Arg#1)),
        equal(2, arity(Arg#1)),
        X is elt(Arg#1, 0),
        equal(X, fac/2-1(X)),
        Y is elt(Arg#1, 1),
        equal(X, -(Y, X)),
        equal(Y, fac/2-1(Y))
        & struct(u, Arg#1, Arg#1).
f/2-1(Arg#1) :-& cons(Arg#1, cons(Arg#1, nil)).
```

4.2.3.4 $\Lambda \rightarrow$ LL

In the final step, all Λ clauses for a predicate are collected into a single LL function definition, where the clauses are connected by a cascade of `if` and `let` statements.[8]

This transformation is only allowed if the following conditions are satisfied:

1. *for deeply deterministic functional predicates*:
   (a) all (REL input) clauses but the last must contain a *quasi-final cut*, i.e. all *guards* (test predicates and test builtins) come before the cut
   (b) the last clause must be a *catch-all* clause, i.e. a clause without any guards
2. *for deeply deterministic test predicates*: all clauses but the last either
   (a) contain a quasi-final cut or
   (b) their head is disjoint from all following clause heads (disjointness is established via non-unifyability)

These conditions ensure that all deeply deterministic functional predicates have *exactly one* solution and all deeply deterministic test predicates have *at most one* solution.

For a *typed* relational language, these conditions could be relaxed in a very desirable way: clauses often do not have to contain *cuts*, which at least by logic-programming purists are disliked for various reasons, and catch-all clauses can in most cases be avoided, since guards together with type declarations can be used

[8] In analogy to PROLOG indexing, for relations consisting of many clauses, `case` statements in addition to the `if/let` cascade could be used.

to prove all clauses of a predicate to be *disjoint* (and *total*) w.r.t. the mode and type declaration.[9]

Example 4.7
In the presence of type declarations, the factorial function can be defined as:

type `fac(`NAT`,`$POSINT$`).`
mode `fac(`$\mathcal{G}$`,`$\mathcal{X}$`).`

```
fac(0,1).
fac(X,Y) :-
  >(X,0),
  X1 is -(X,1),
  fac(X1, FX1),
  Y is *(X, FX1).
```

Now both clauses are disjoint and together catch all cases *w.r.t. the type declaration.* △

In example 4.6, both predicates satisfy the conditions, i.e. clauses `1a` and `2a` contain final cuts and clauses `1b` and `2b` are catch-all clauses. The following LL function definitions result:

```
(defun fac/2-1 (arg#1)
  (if (equal 0 arg#1)
      1
      (* arg#1 (fac/2-1 (- arg#1 1)))))

(defun f/2-1 (arg#1)
  (if (and (structp arg#1)
           (equal 's (functor arg#1))
           (equal 2 (arity arg#1)))
      (let ((x (elt arg#1 0)))
           (if (equal x (fac/2-1 x))
               (let ((y (elt arg#1 1)))
                    (if (and (equal x (- y x)) (equal y (fac/2-1 y)))
                        (struct 'u arg#1 arg#1)
                        (cons arg#1 (cons arg#1 nil))))
               (cons arg#1 (cons arg#1 nil))))
      (cons arg#1 (cons arg#1 nil))))
```

[9] In special cases, disjointness can be established by global analysis even for programs without type declarations [Debray and Warren, 1990].

In case of the `fac` function, exactly the definition a human programmer would have chosen has been created.

The function definition for `f` is extremely *operational*: the sequence of unifications of the original REL definition has been transformed into a very precise operational definition mainly consisting of assignments and equality tests. This is consistent with recent insights that PROLOG programs should be compiled into much simpler and more specialized instructions than the WAM instructions (see [Taylor, 1990; Van Roy, 1994]).

The three copies of `(cons arg#1 (cons arg#1 nil))` cannot be avoided in a purely functional specification. These copies, which do not cause any efficiency disadvantages, can be removed on the abstract machine level by code sharing.

4.2.3.5 Examples

In this subsection, some of the features of the determinism analysis and transformation that were not covered by example 4.6 are described with the help of some additional examples:

- `even`: a simple deeply deterministic *test* predicate
- `append` and `reverse`: functional predicates showing how *free variables* can be handled in LL and how to avoid the specification of *catch-all clauses*

Example 4.6 only contained two deeply deterministic *functional* predicates. The following example shows how *test* predicates are compiled.

Example 4.8

```
even(0) :- !.
even(X) :- >(X, 1), even(-(X,2)) !.
even(X) :- <(X,-1), even(+(X,2)).
```

`even` is a deeply deterministic test predicate w.r.t. the mode declaration `even(`$\mathcal{G}$`)`. It relies on the closed-world assumption that (after possible recursions) `even(1)` and `even(-1)` yield 'no' because no clauses cover these cases. △

`even` is compiled into the following LL function definition:

```
(defun even (arg#1)
  (if (equal 0 arg#1)
      t
      (if (and (> arg#1 1) (even (- arg#1 2)))
          t
          (if (and (< arg#1 -1) (even (+ arg#1 2)))
              t
              nil))))
```

The transformation process is nearly identical to the transformation for functional predicates with the following exceptions:

1. the transformation into RELFUN functions is not necessary
2. in $(\mathcal{E}, v)$, and thus in Λ, the return value (v) is always `t`
3. the `if` and `let` cascade created by $\Lambda \rightarrow$ LL closes with `nil`; here, the final `nil` explicitly covers the cases `arg#1=1` and `arg#1=-1`

The next example, `append` and `reverse`, is used to show

- how to avoid the specification of *catch-all clauses* and
- how *free variables* can be handled in LL.

Example 4.9

```
append([], X, X) :- !.
append([H|T], X, [H|Y]) :- append(T,X,Y) !.
append(X,Y,non-list-arg).

reverse([],[]) :- !.
reverse([H|T],X) :- reverse(T,T1), append(T1,[H],X) !.
reverse(X,non-list-arg).
```

`append` and `reverse` are deeply deterministic functional predicates w.r.t. the mode declarations `append(`$\mathcal{G}$`,`$\mathcal{G}$`,`$\mathcal{X}$`)` and `reverse(`$\mathcal{G}$`,`$\mathcal{X}$`)`. $\triangle$

Since in untyped PROLOG and RELFUN[10] there is no way to specify that `append` and `reverse` only operate on lists, the weird catch-all clauses and additional cuts are needed.

In order to avoid these and all cuts, it is possible to *declare* predicates to be total deterministic functional predicates. With such declarations[11], `append` and `reverse` can be specified as one is used to in PROLOG, which leaves it to the user to call them with the intended types:

```
append([], X, X).
append([H|T], X, [H|Y]) :- append(T,X,Y).

reverse([],[]).
reverse([H|T],X) :- reverse(T,T1), append(T1,[H],X).
```

The following LL function definitions result:

[10] The extension of RELFUN by type definitions and signatures is momentarily worked on.

[11] The syntax of these declarations is described in appendix B.

```
(defun append/3-1 (arg#1 arg#2)
  (if (equal nil arg#1)
      arg#2
      (if (consp arg#1)
          (cons (car arg#1) (append/3-1 (cdr arg#1) arg#2))
          (type-error))))

(defun reverse/2-1 (arg#1)
  (if (equal nil arg#1)
      nil
      (if (consp arg#1)
          (append/3-1 (reverse/2-1 (cdr arg#1)) (cons (car arg#1) nil))
          (type-error))))
```

Since no catch-all clauses were specified, `(type-error)` is generated. Furthermore, the resulting LL definitions are slightly different from the definitions a human programmer would have chosen: since the compiler does not know that `append` and `reverse` only operate on lists, type-checking code is generated: `(consp arg#1)`.

In the design of LL, (free) *logical variables* were not taken into account. In our implementation, they are simply ignored: they cannot be accessed in LL functions, but lists and structures with unbound variables that were created in REL can be used as arguments for LL functions. From LL's point of view, free variables belong to a data type for which neither constructors nor selectors exist.

It is thus possible to append or reverse lists with the above LL functions even if they contain free variables. In the further execution of the embedding REL program, these free variables can be bound: the query

```
X is [a,b,Y,f[Z]], reverse(X,XR), Y is c, Z is d
```

yields the following bindings:

```
X = [a, b, c, f[d]]
XR = [f[d], c, b, a]
Y = c
Z = d
```

4.3 Implementing Extra-logicals via LL

A very flexible way of introducing extensions in relational languages is by specifying them in LL.[12] In the following, a simple REL extension is described: *global variables*.

Global variables can be implemented by allowing the following two LL functions, `setvar` and `getvar`, to be accessed from REL:

```
(defun setvar (var value)
  (eval (list 'setq var (list 'quote value))))

(defun getvar (var)
  (eval var))
```

(`setvar` *var value*) sets the global LL variable *var* to *value* by evaluating (`setq` *var* '*value*). (`getvar` *var*) returns the value of the global LL variable *var* by simply evaluating *var*.

From the point of view of REL, such a variable is a 'constant' changeably associated with a term.

Global variables are useful in REL in two situations:

1. when computed values have to live longer than a query (this is usually the case for programs with user interaction and/or operating system access)

2. when computed values have to survive backtracking

The second situation occurs when all solutions of a query have to be collected: `bagof` in PROLOG or `tupof` in RELFUN.

In pure PROLOG and RELFUN, it is impossible to collect all solutions of a query: intermediate results are reset when backtracking occurs, which is used to generate the next solution; thus all computed values are lost. In order to overcome this difficulty, `bagof` in PROLOG and `tupof` in RELFUN were introduced.

These extra-logicals can easily be implemented via global variables: a global variable holds a stack of lists containing the intermediate results (a *stack* is necessary because it is possible that in the execution of a `tupof` another `tupof` call occurs).

[12] Since our prototypical implementation does not (yet) allow LL functions to access (bind) free REL variables, no extensions of the unification can be accomplished.

Chapter 5

LLAMA — The LISP light Abstract Machine

The LLAMA is a simple and universal abstract stack machine [Henderson, 1980] which is used as the target machine for the compilation of LL. It is universal in the sense that most functional[1] and many imperative language can easily be compiled into it.

In the following sections, the constituents of the LLAMA,

- the registers and memory organization,
- the representation of data structures,
- the calling conventions, and
- the instructions

are described.

5.1 Registers and Memory Organization

The LLAMA was designed to have as few registers as possible.

LLAMA registers	
P	program counter
SP	stack pointer
H	heap pointer
CT	catch stack pointer

P, SP, and H are shared with the WAM. Only the CT register is new; it is used for non-local exits (`catch` and `throw`) and loops (`loop` and `do`) which are implemented via non-local exits. Programs not using these builtins never access the CT register, thus no overhead for purely functional programs results.

[1] With eager (call-by-value) evaluation

The LLAMA has the following memory organization:

LLAMA memory organization	
Stack	data and return address stack
Heap	heap area for complex data structures
Catch Stack	catch stack for non-local exits
Code	code area
Memory	general purpose memory

Stack, Heap, and Code are shared with the WAM. Catch Stack is used to store tuples of the form ⟨*tag stackpointer address*⟩ which are created by `catch` and removed by `throw`.

Memory is used to store data structures which have to live longer than entries in Stack and Heap which are reset between queries. Thus, Memory is mainly used to store global variables. Furthermore, Memory contains hash and jump tables which are created by the GAMA (see section 7).

5.2 The Representation of Data Structures

The LLAMA represents data structures exactly like the WAM. This allows LL functions to access and manipulate data that has been created by REL programs without having the need of copying.

In the WAM (and thus in the LLAMA), Stack and Heap have a tagged architecture: an entry consists of a tuple ⟨*tag value*⟩. The following sections describe the representation of constants, lists, and structures on Stack and Heap.

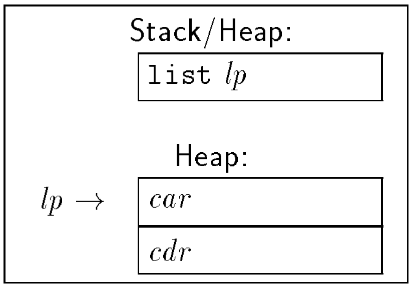
5.2.1 Constants

A constant c is the only data structure which is simple enough to fit into a single stack or heap cell:

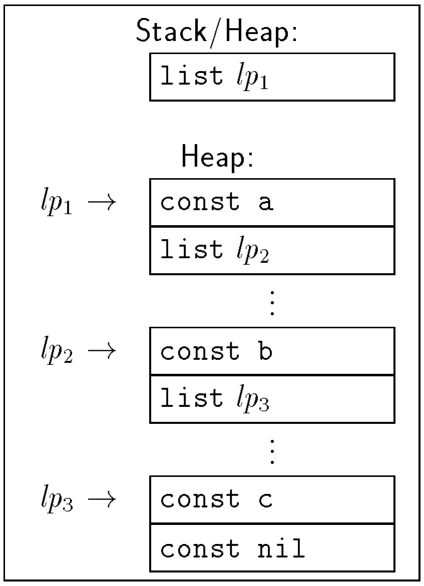
Stack/Heap:
`const` c

5.2.2 Lists

Lists are — just as in COMMON LISP — handled by (“`cons`”) pairs. Such a `cons` pair (car . cdr) is represented as follows:

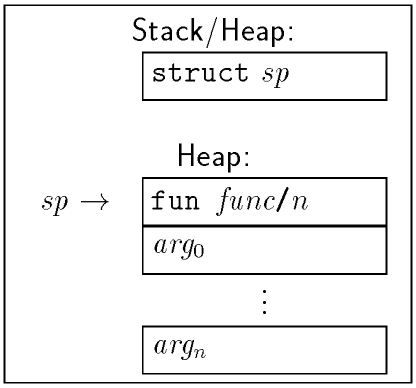


The list `(a b c)` is equivalent to `(a . (b . (c . nil)))`:



5.2.3 Structures

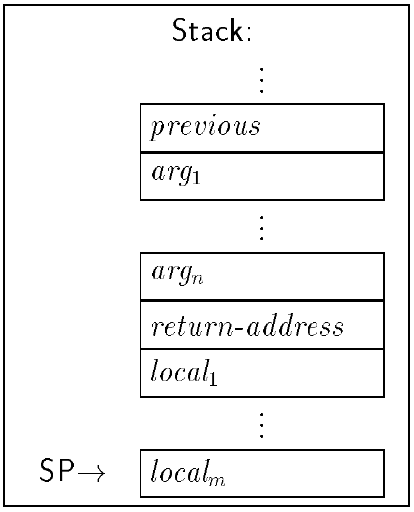
Structures are represented similarly to lists:



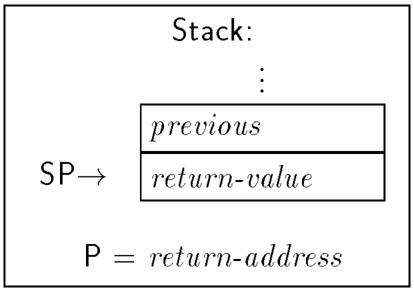
5.3 Calling Conventions

As in all stack languages, the arguments are put on the stack before a function is executed.

In case of a user-defined function, the return address is additionally put on the stack; local variables (e.g. created by `let`) follow:



After executing the function, all arguments, the return address, and local variables are removed from the stack, the return value is put on the stack, and the program counter is set to the return address:



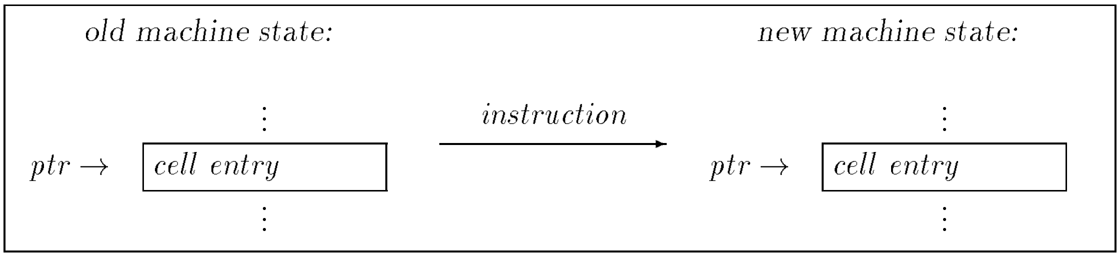
5.4 The Instructions

In the following sections, a detailed description of all LLAMA instructions is given. The instructions are defined via a simple and uniform graphical representation which allows an easy re-implementation in any (functional or imperative) language.

Alternatively, the definitions could have been given using an imperative pseudo language as in

push-constant c		
SP	$\leftarrow$	SP $\oplus$ 1
mem(SP)	$\leftarrow$	$\langle$const $c\rangle$

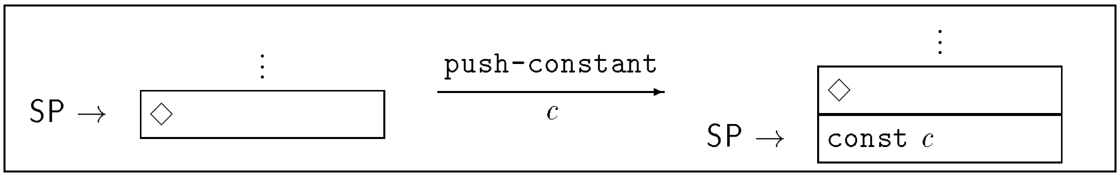
Even for slightly more complex instructions, this representation becomes unreadable. Therefore, diagrams of the following form are used:



The LLAMA instruction set contains the following instructions:

LLAMA instruction set	
constants	`push-constant`
lists	`cons`, `car`, `cdr`, `null`, `consp`
structures	`struct`, `structp`, `functor`, `arity`, `elt`
equality	`eq`, `eql`, `equal`
stack manipulation	`pop`, `remove`, `dup`, `set-nth`, `set-nth-`
control instructions	`ll-call`, `return`, `goto`, `on-nil-goto`,
	`catch`, `remove-tag`, `throw`, `catch-nil`, `throw-nil`
higher-order instructions	`funcall`, `apply`, `eval`
global variables	`set-global`, `set-global-`, `push-global`
numerical/string builtins	`ll-builtin`
I/O builtins	`read`, `print`

5.4.1 Constants



In order to mark the place of cells whose contents is irrelevant, cells containing a ◇ or □ are used.

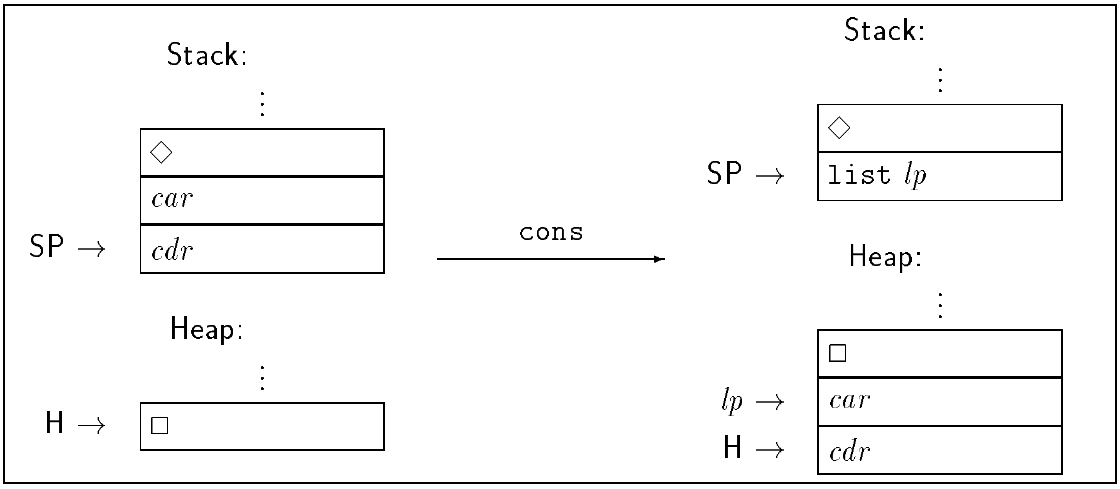
5.4.2 Lists

The operations on lists are divided into three classes:

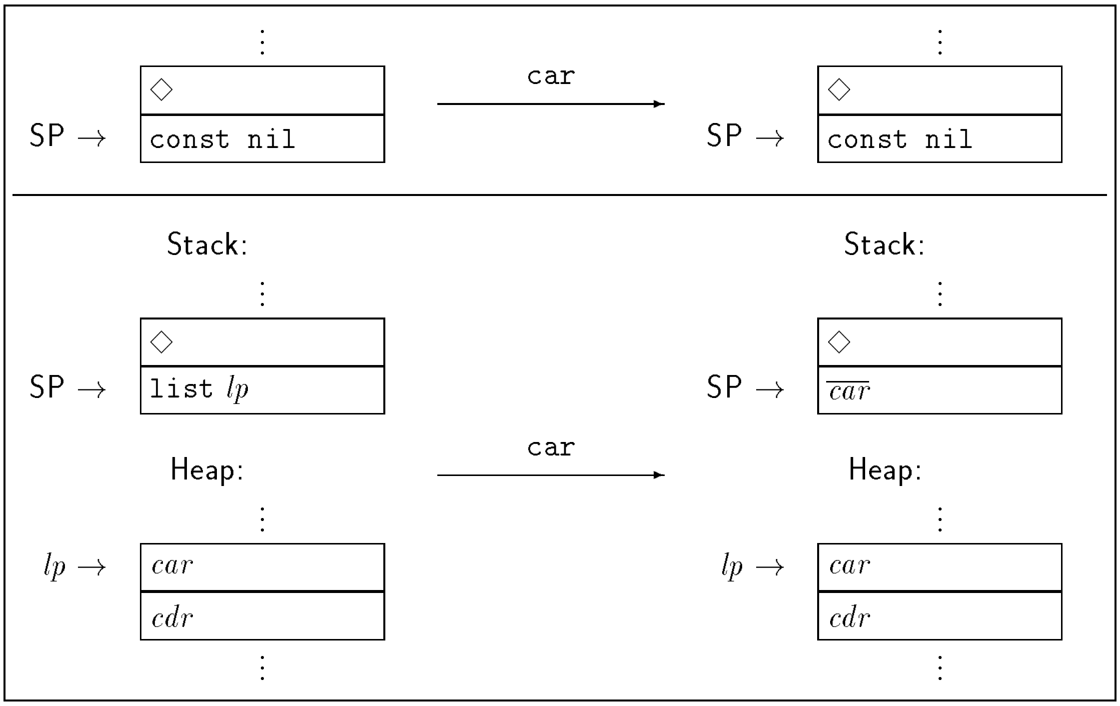
1. *construction*: `cons`
   `cons` creates a `cons` pair; general lists are first transformed into `cons` lists, as described in section 6.2.2

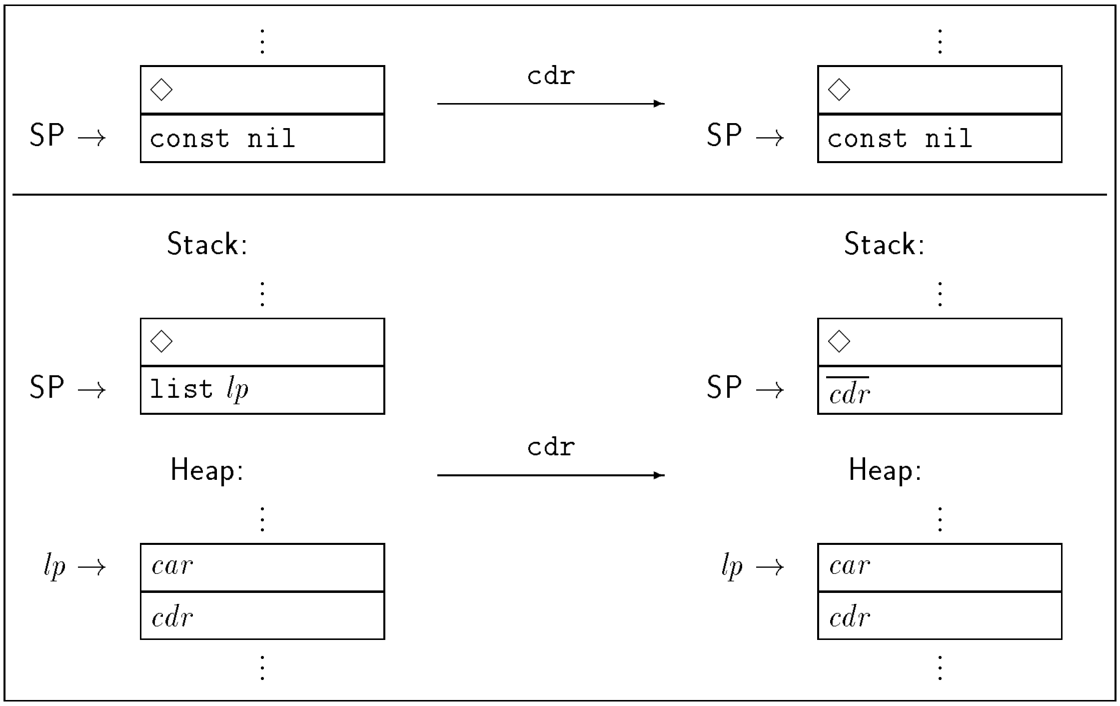
2. *selection*: `car`, `cdr`
   with `car` and `cdr`, `cons` pairs are taken apart; by successive application of `cdr`, any list element can be accessed

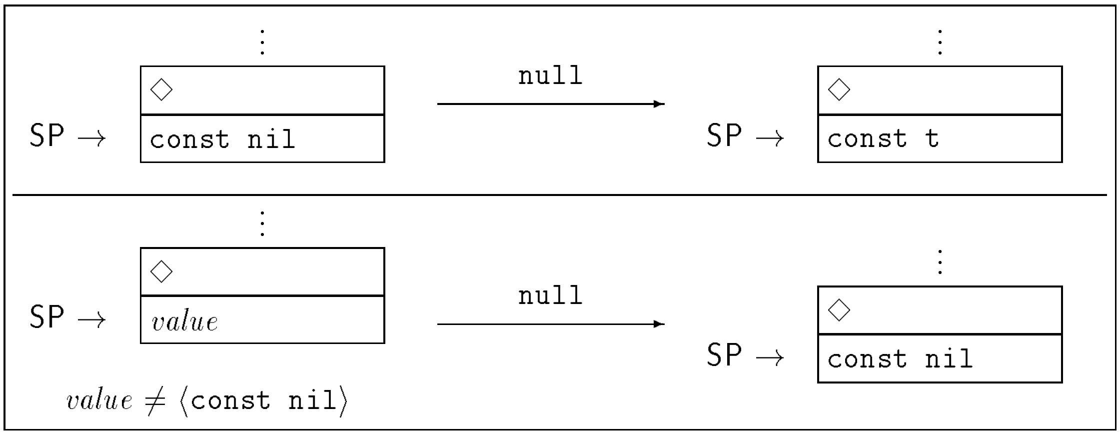
3. *test*: `null`, `consp`
   `null` tests for an empty list (`nil`), `consp` for a non-empty list (a `cons` pair)

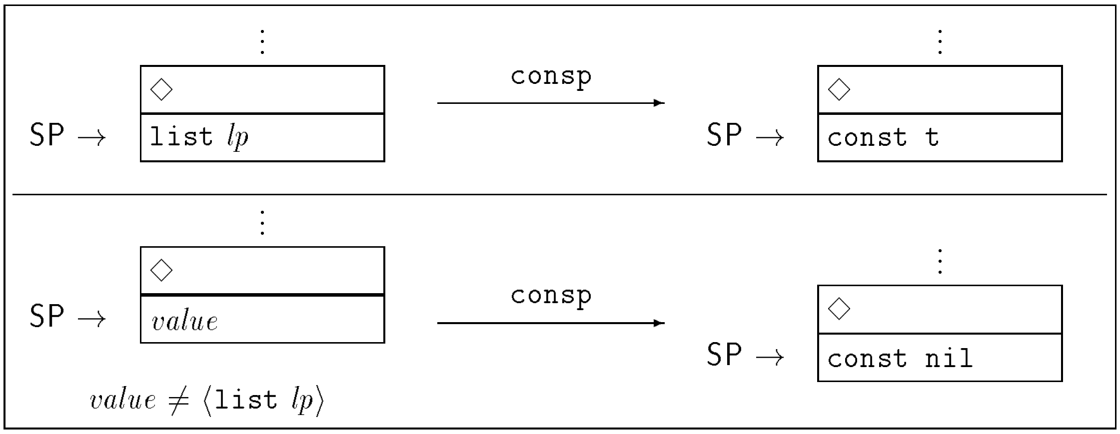


For the access functions `car` and `cdr`, the accessed heap entries have to be dereferenced because they could have been created by REL in the context of free variables (which were bound later) and then passed to LL. Entries put on the stack have always to be dereferenced in order to avoid unnecessary dereferencing when multiply accessing arguments or local variables. Dereferencing is denoted by $\overline{arg}$.


SP →
◇
const nil
car
SP →
◇
const nil
Stack:
SP →
◇
list lp
Heap:
lp →
car
cdr
car
Stack:
SP →
◇
car
Heap:
lp →
car
cdr


SP →
◇
const nil
cdr
SP →
◇
const nil
Stack:
SP →
◇
list lp
Heap:
lp →
car
cdr
cdr
Stack:
SP →
◇
cdr
Heap:
lp →
car
cdr





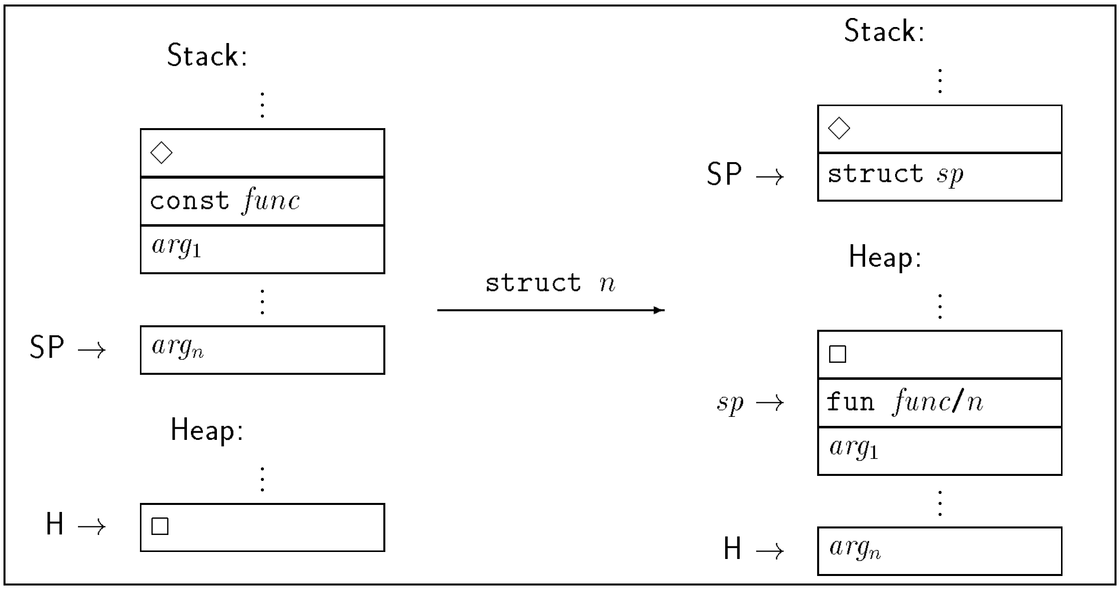
5.4.3 Structures

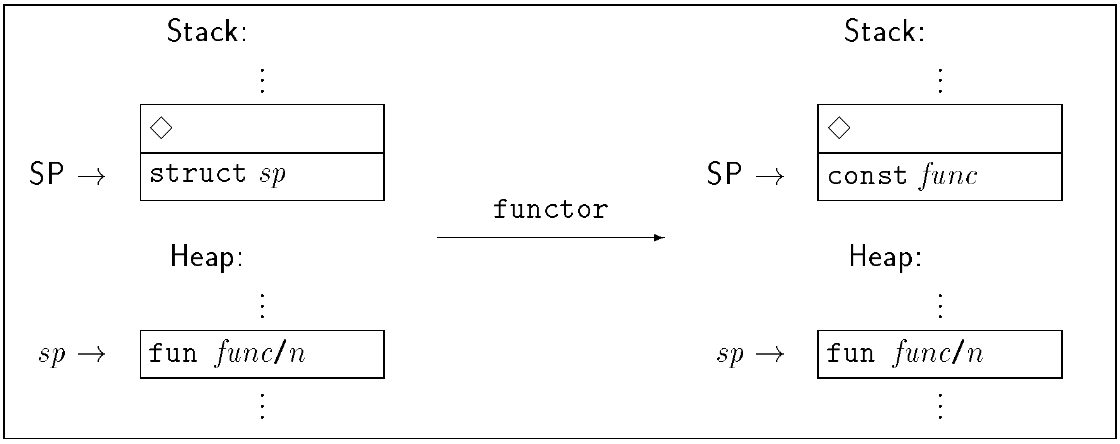
In analogy to lists, the following classes of operations on structures exist:

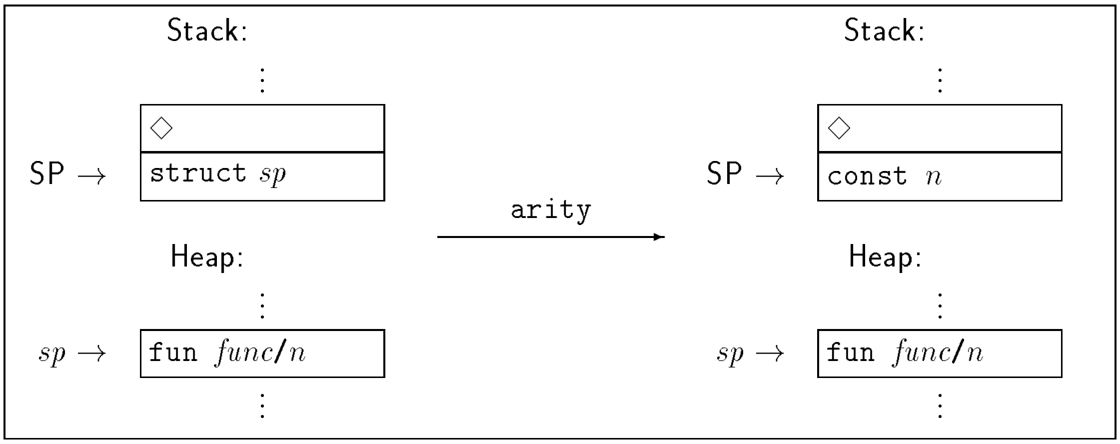
1. *construction*: `struct` n
   `struct` n creates a structure with arity n; functor and arguments are on the stack

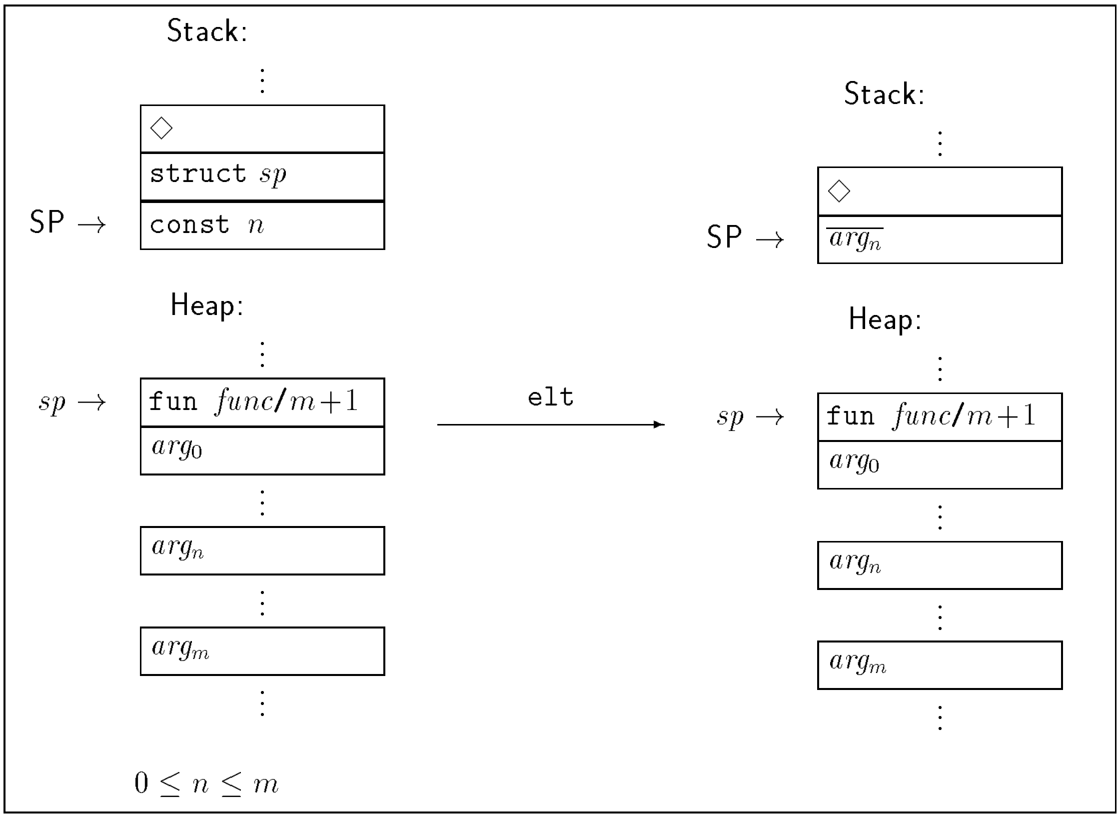
2. *selection*: `functor`, `arity`, `elt`
   with `functor` and `arity`, the functor and the arity of a structure are determined; `elt` expects a structure and a number n on the stack and determines the n^{th} argument of the structure, where the first argument is counted as the 0^{th} argument

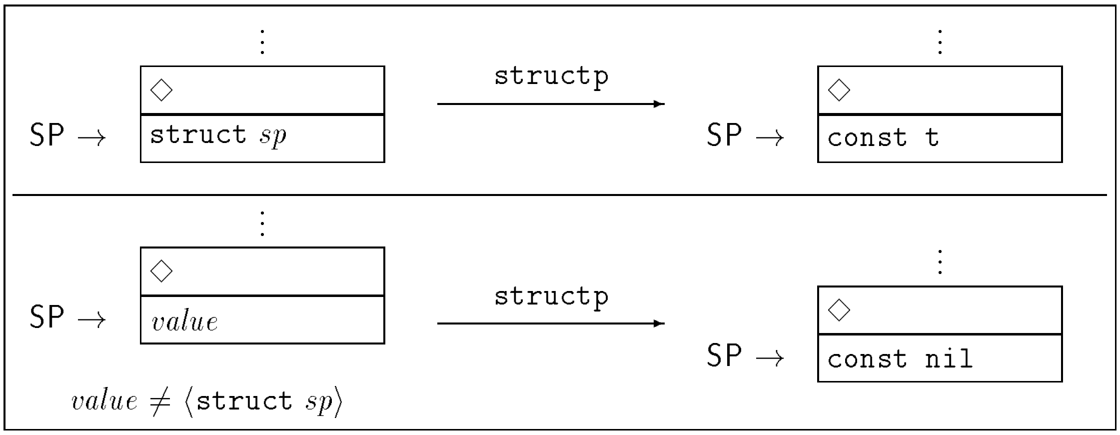
3. *test*: `structp`
   `structp` tests for a structure


Stack:
◇
const func
arg1
SP →
argn
Heap:
H →
□
struct n
Stack:
◇
SP →
struct sp
Heap:
□
sp →
fun func/n
arg1
H →
argn


Stack:
◇
SP →
struct sp
Heap:
sp →
fun func/n
functor
Stack:
◇
SP →
const func
Heap:
sp →
fun func/n


Stack:
◇
SP →
struct sp
Heap:
sp →
fun func/n
arity
Stack:
◇
SP →
const n
Heap:
sp →
fun func/n

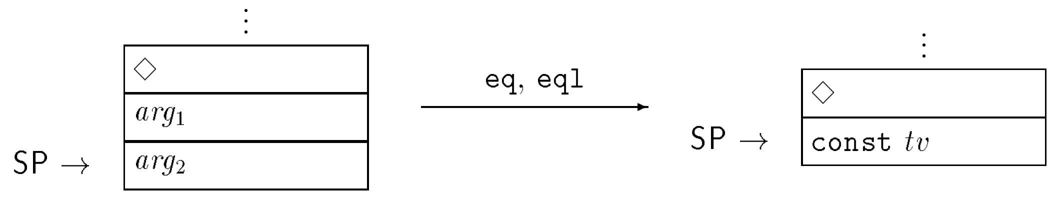




5.4.4 Equality

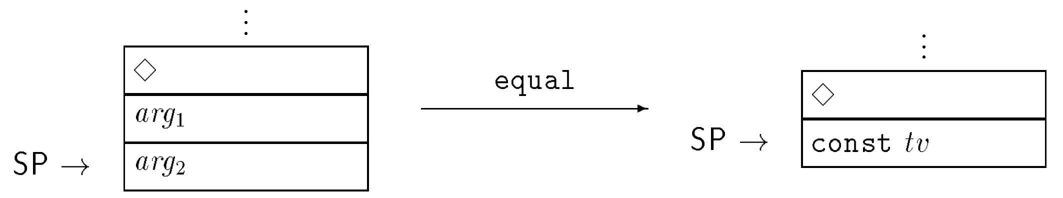
In LL, just as in COMMON LISP, there are two forms of equality: *structural* equality and *toplevel* equality.

Toplevel equality (`eq` and `eql`) tests, if the representation of two arguments is equal at the toplevel, i.e. only pointers (in the case of lists and structures) are compared. This form of equality was only provided for compatibility with COMMON LISP; its use is not recommended.



$$tv = \begin{cases} \texttt{t} & , arg_1 = arg_2 \\ \texttt{nil} & , \text{otherwise} \end{cases}$$

Structural equality (`equal`) tests, if the two arguments have the same structure on stack and heap, i.e. if their external term representation is equal.



$$tv = \begin{cases} \texttt{t} & , arg_1 \doteq arg_2 \\ \texttt{nil} & , \text{otherwise} \end{cases}$$

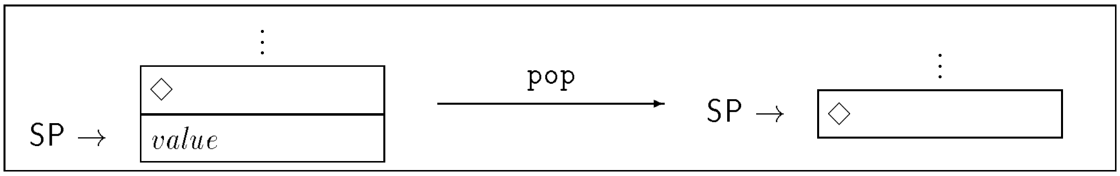
The structural equality $\doteq$ is defined as follows:

$$\begin{array}{lll} x \doteq y & \iff & x = y \\ & \vee\ (& x = \langle \texttt{list}\ lp_x \rangle \wedge y = \langle \texttt{list}\ lp_y \rangle \\ & & \wedge\ \overline{\mathrm{mem}(lp_x)} \doteq \overline{\mathrm{mem}(lp_y)} \\ & & \wedge\ \overline{\mathrm{mem}(lp_x \oplus 1)} \doteq \overline{\mathrm{mem}(lp_y \oplus 1)} \quad)^2 \\ & \vee\ (& x = \langle \texttt{struct}\ sp_x \rangle \wedge y = \langle \texttt{struct}\ sp_y \rangle \\ & & \wedge\ \mathrm{mem}(sp_x) = \langle \texttt{fun}\ fun/n \rangle \\ & & \wedge\ \mathrm{mem}(sp_y) = \langle \texttt{fun}\ fun/n \rangle \\ & & \wedge\ \forall_{i \in \{1,\ldots,n\}}\ \overline{\mathrm{mem}(sp_x \oplus i)} \doteq \overline{\mathrm{mem}(sp_y \oplus i)} \quad) \end{array}$$

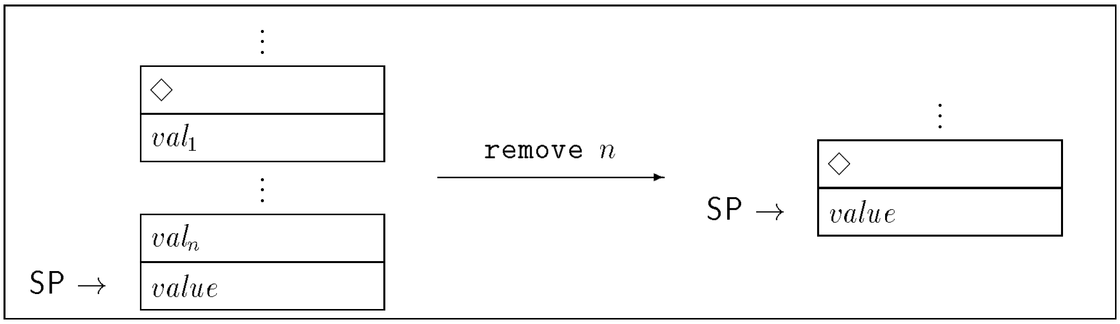
5.4.5 Stack Manipulation

`pop` simply removes the top stack element. This instruction is only used in non-functional programs, e.g. in loops, where the result of the loop body is simply discarded.

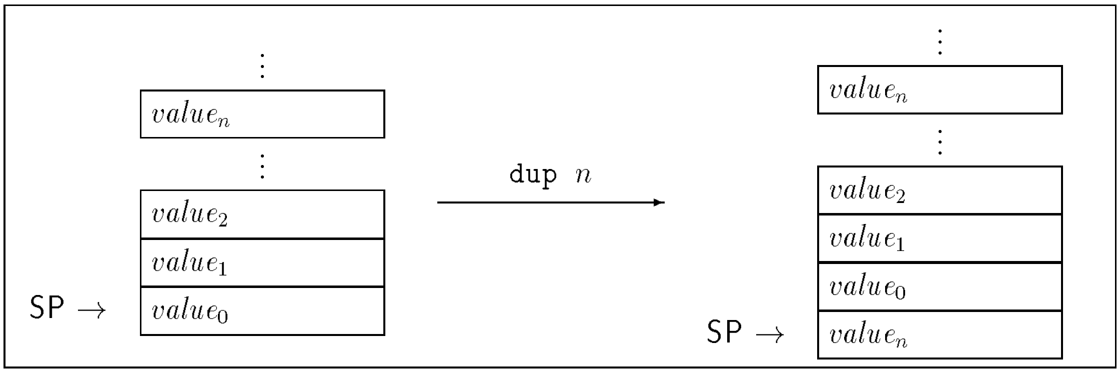
[2] $p \oplus n$ means incrementing the pointer p by n cells.



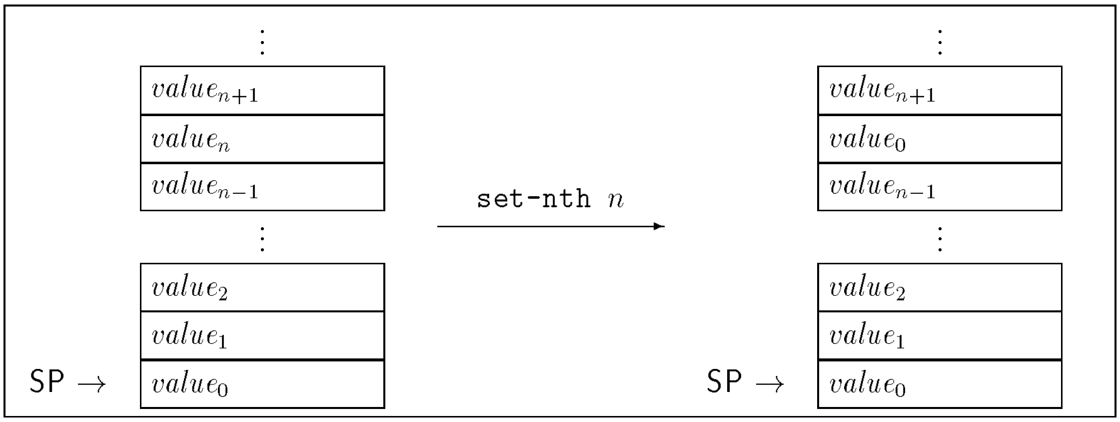
`remove` removes n elements from the stack, except the topmost one. This instruction is used at the end of a `let` statement where the local variables have to be discarded.

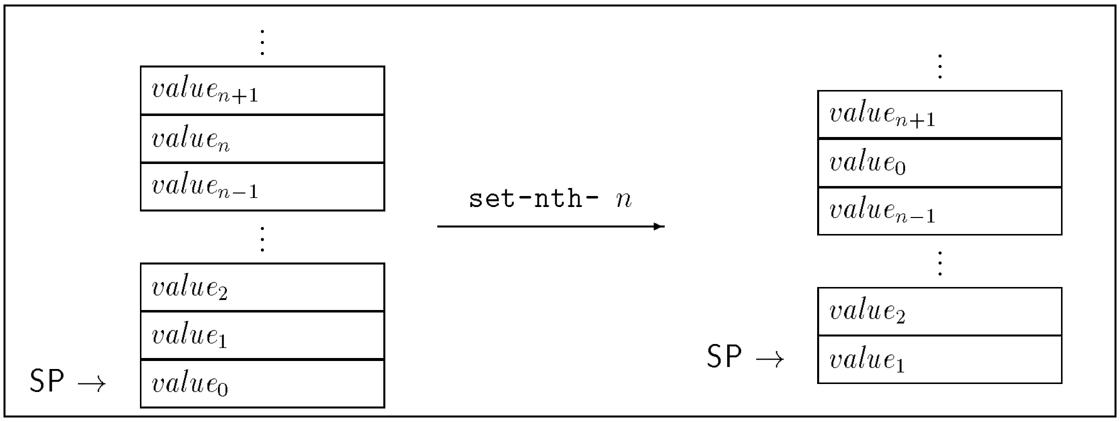


`dup` duplicates the n^{th} element of the stack. This is used to push a function argument or a local variable (as an operand) on the stack.



`set-nth` replaces the n^{th} element with the top stack element. This instruction is used for assignments (e.g. via `setq`). The `set-nth-` instruction additionally removes the top stack element (see section 6.3).


$value_{n+1}$
$value_n$
$value_{n-1}$
$value_2$
$value_1$
SP →
$value_0$
set-nth n
$value_{n+1}$
$value_0$
$value_{n-1}$
$value_2$
$value_1$
SP →
$value_0$


$value_{n+1}$
$value_n$
$value_{n-1}$
$value_2$
$value_1$
SP →
$value_0$
set-nth- n
$value_{n+1}$
$value_0$
$value_{n-1}$
$value_2$
SP →
$value_1$

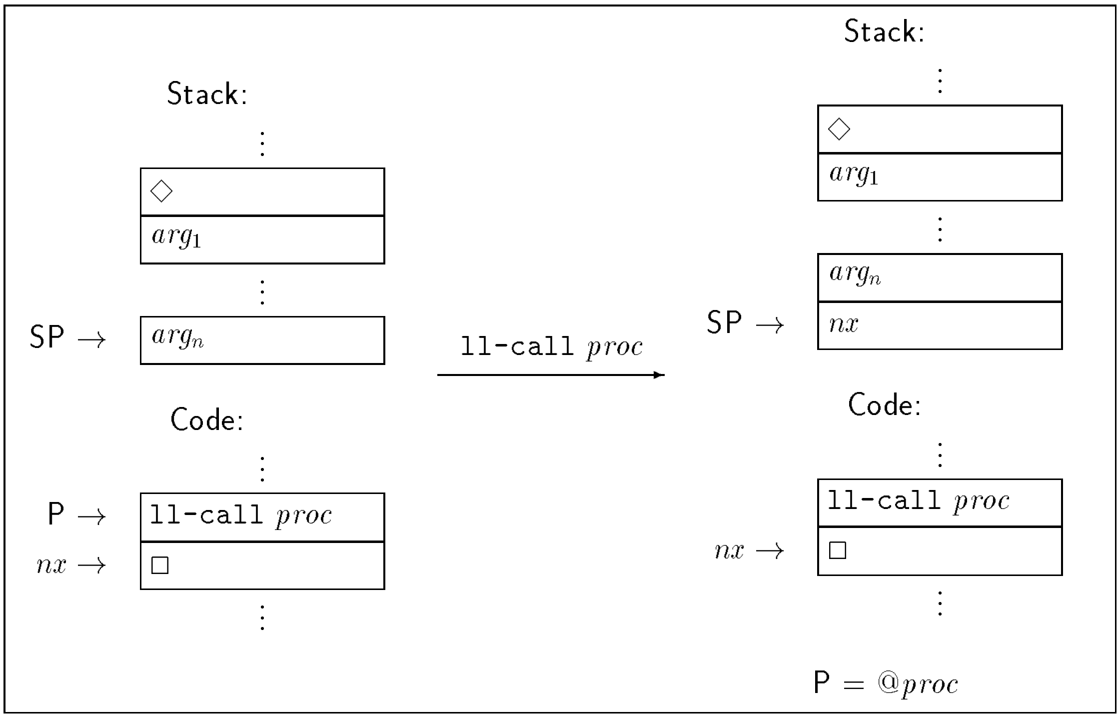
5.4.6 Control Instructions

The control instructions are divided into three classes:

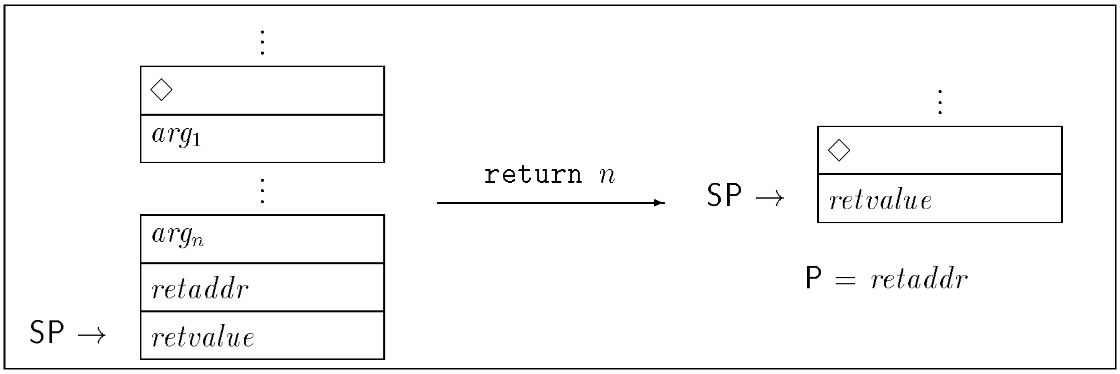
1. *calling subroutines*: `ll-call`, `return`
2. *branch instructions*: `goto`, `on-nil-goto`
3. *non-local exits*: `catch`, `remove-tag`, `throw`, `catch-nil`, `throw-nil`

5.4.6.1 Calling Subroutines

`ll-call` *proc* pushes the return address on the stack and stores the address of *proc* (denoted as @*proc*) in P.

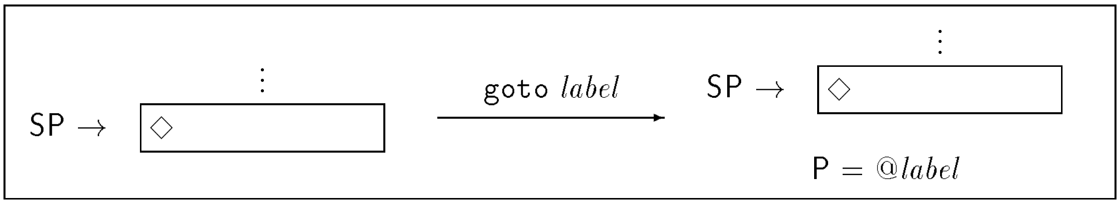


`return` n removes the n arguments and the return address *retaddr* from the stack and continues execution at P = *retaddr*.

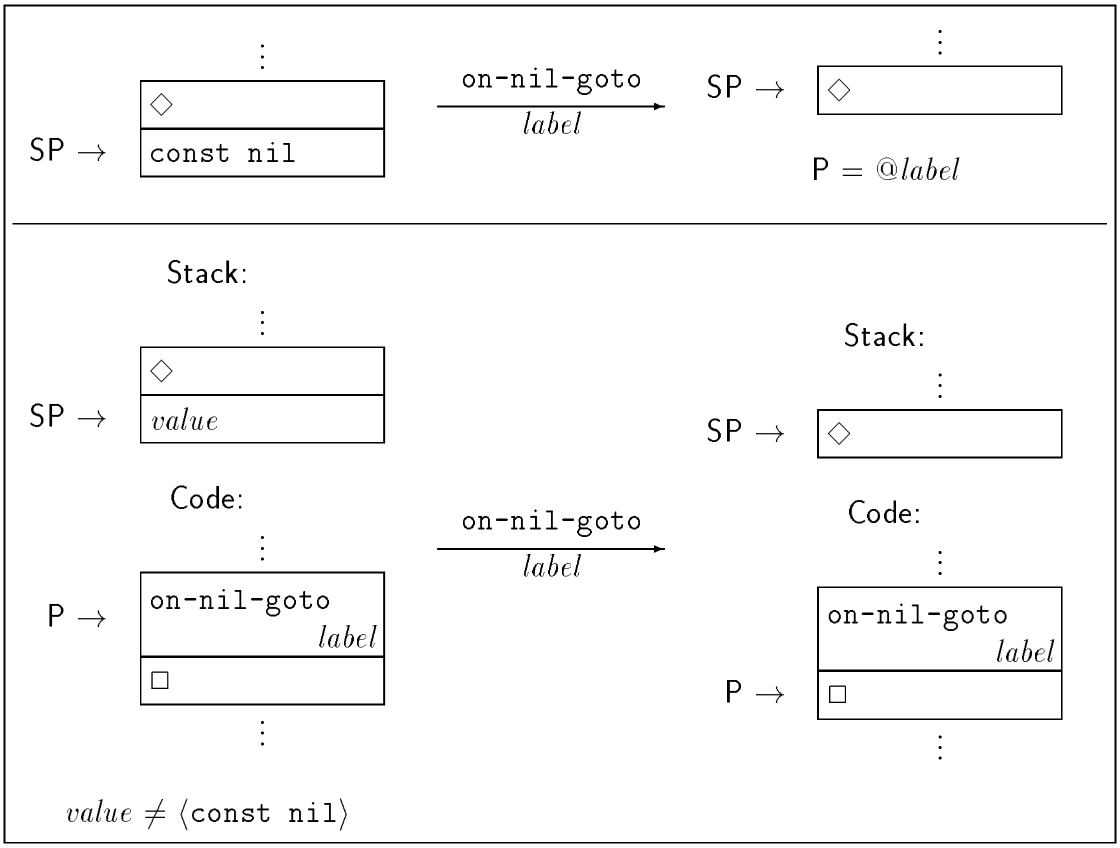


5.4.6.2 Branch Instructions

`goto` *label* is the *unconditional* branch instruction: execution is continued at P = @*label*.



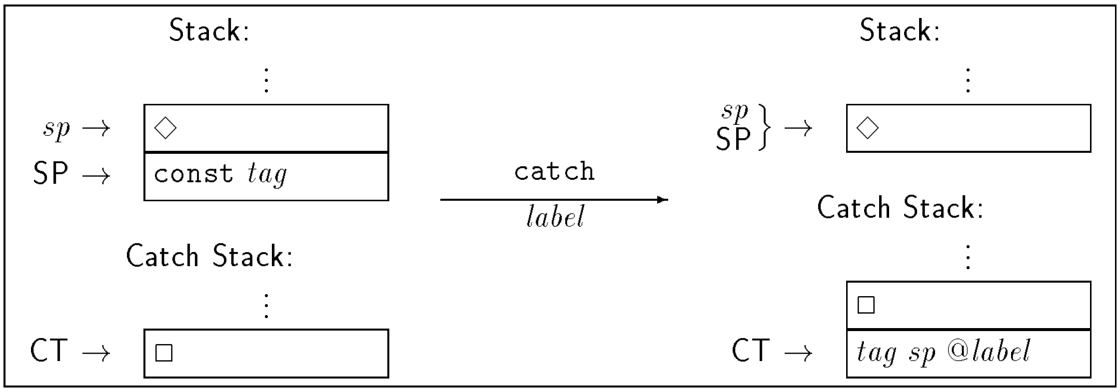
`on-nil-goto` *label* is the only *conditional* branch instruction in the LLAMA: execution is continued at P = @*label* iff the top stack element is `nil`; execution continues at the instruction following the `on-nil-goto` instruction otherwise.



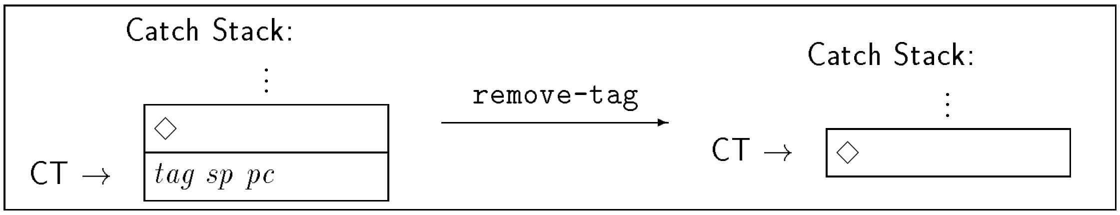
5.4.6.3 Non-local Exits

`catch` *label* prepares the catch stack at the beginning of an LL `(catch` *tag* `.` *body*`)` statement: the actual stack pointer and the address of the instruction following *body* (= @*label*) are pushed on the catch stack.

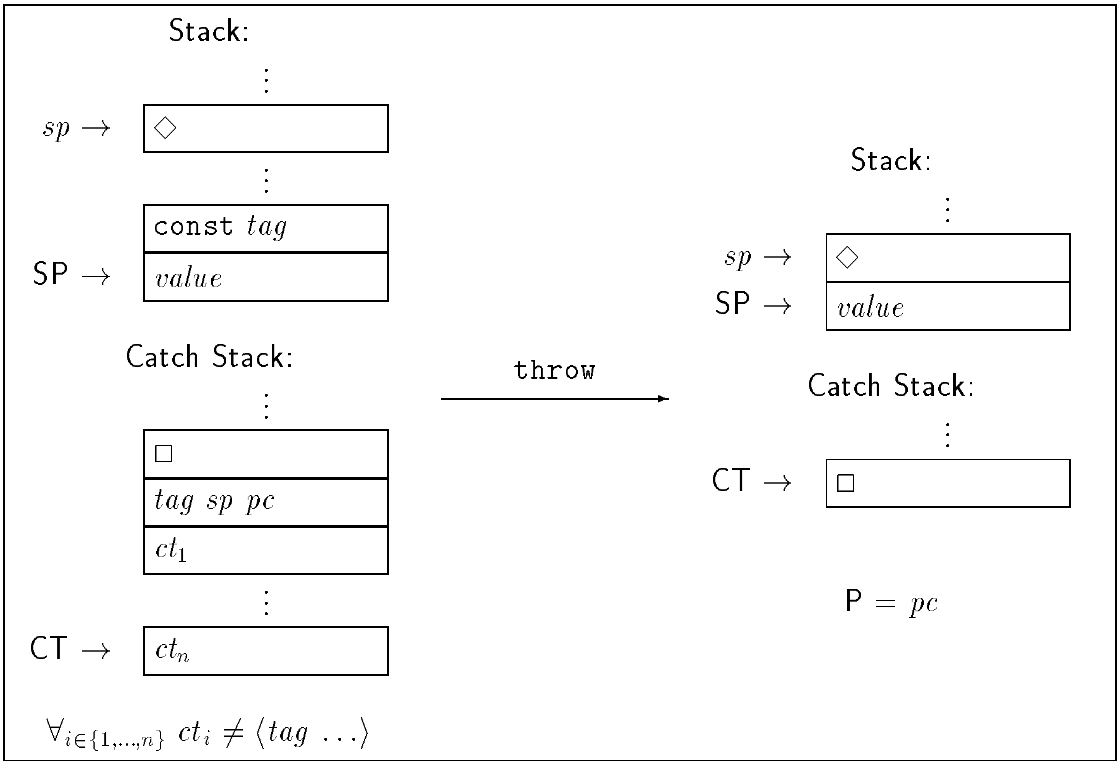
For a detailed definition of the compilation of the `(catch` *tag* `.` *body*`)` statement into the `catch` and `remove-tag` instructions, please refer to section 6.2.8.



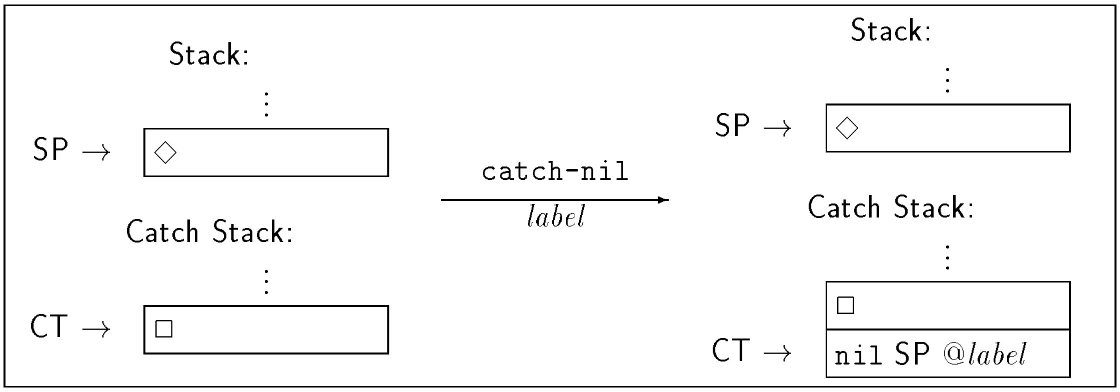
If the end of the body of the `(catch` *tag* `.` *body*`)` statement is reached, the top of the catch stack is removed with the `remove-tag` instruction.

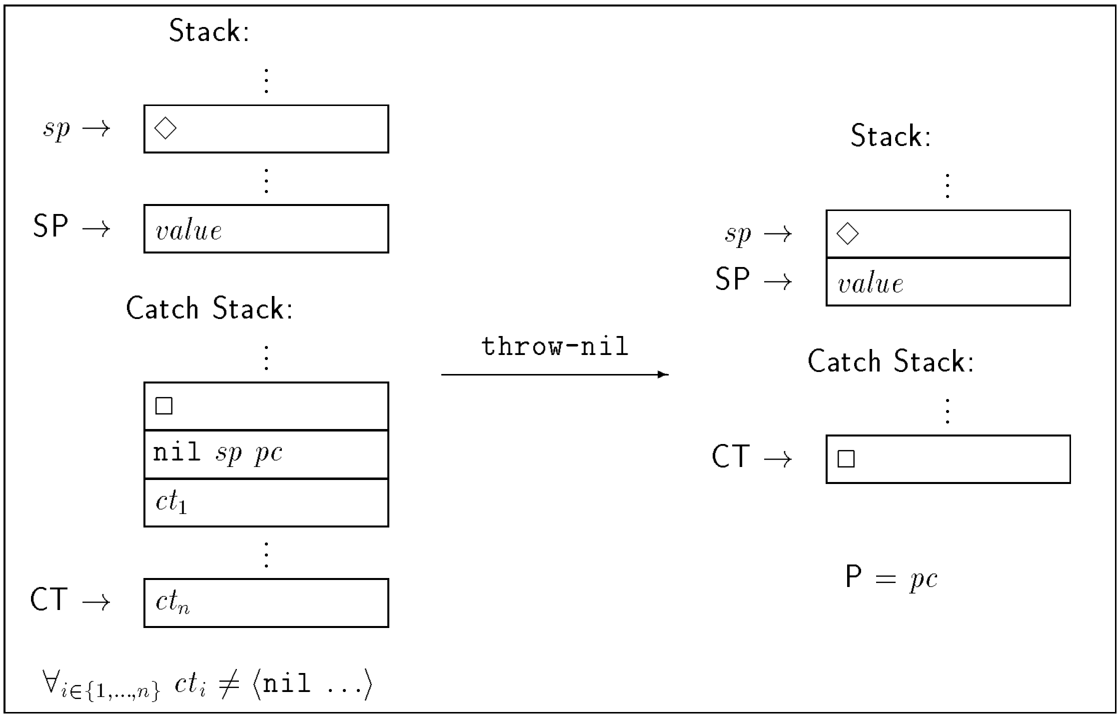


A non-local exit of the body is performed by the `throw` instruction, corresponding to the LL `(throw` *tag value*`)` builtin: the topmost occurrence of the tag in the catch stack is determined; P and SP are restored with the corresponding values in the catch stack, and the found entry and all entries on the catch stack behind it are discarded. It is an error when the tag is not found in the catch stack.



For LL loops (`loop` and `do`), specialized versions of `catch` and `throw` are used where the tag is `nil`. In order to avoid to push the constant `nil` on the stack, a `catch-nil` and a `throw-nil` instruction were introduced.





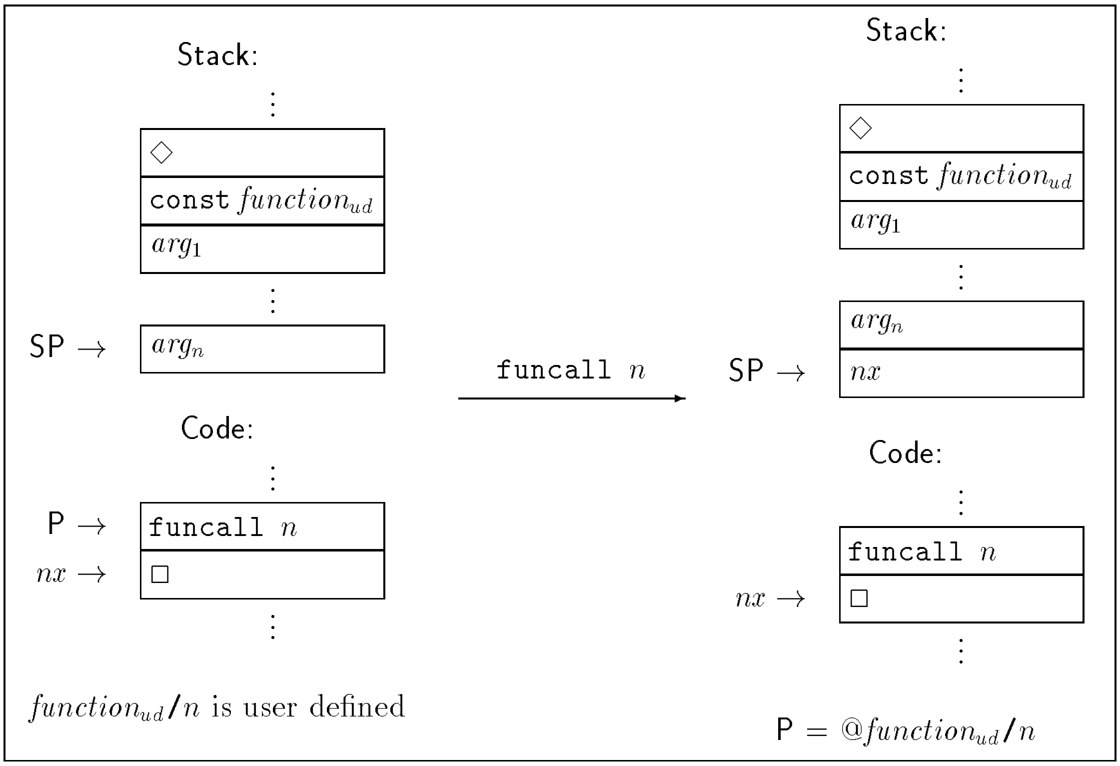
5.4.7 Higher-Order Instructions

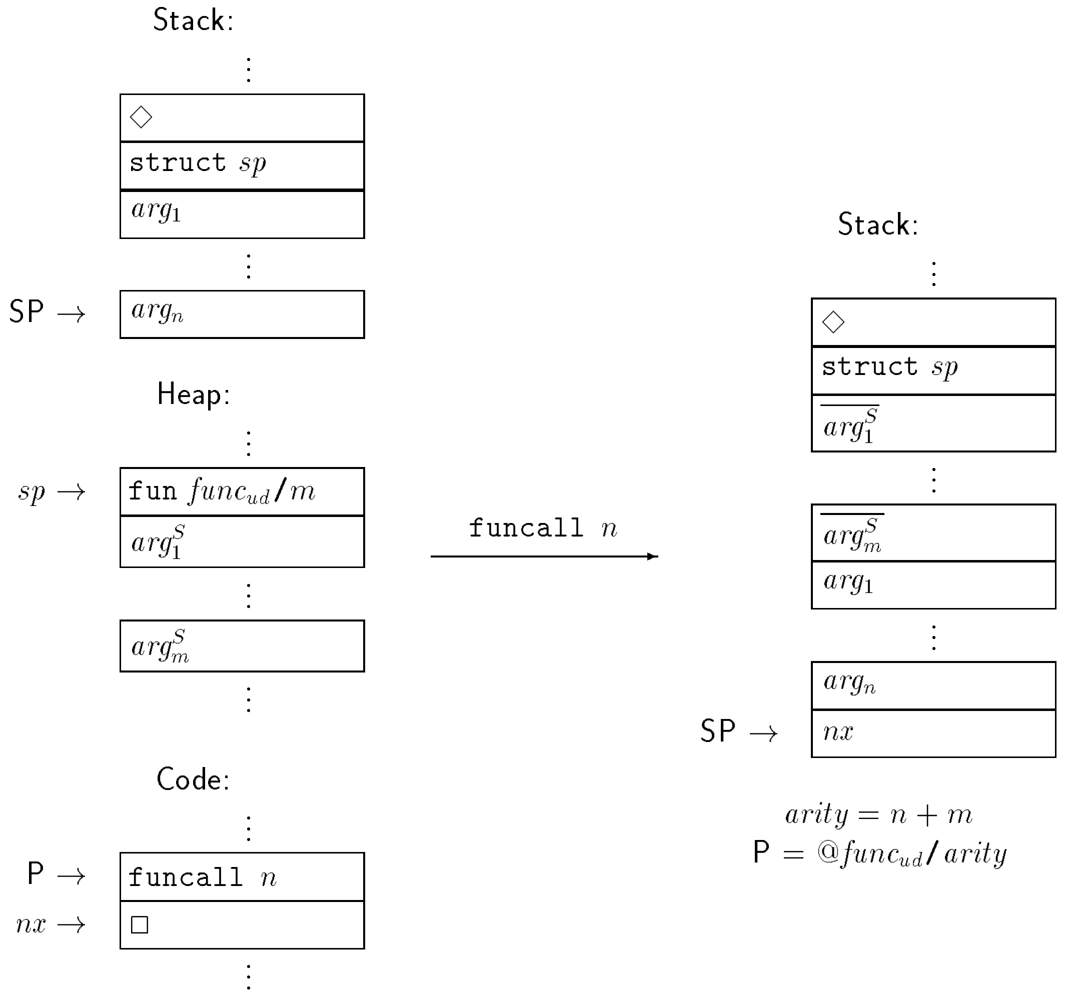
In LL, three builtins for higher-order function application exist: `funcall`, `apply`, and `eval`. Their corresponding LLAMA instructions `funcall` n, `apply` n, and `eval` work as follows:

1. `funcall` n:
   - First, the functional object is examined: if it is a constant *function*, *function*/n is the function to be called. If it is a structure `[`*func* arg_1^S ... arg_m^S`]`, then *func*/*arity* with *arity* = $n + m$ is the function. Structures as functional objects are mainly used for lambda expressions: the free local variables of a lambda expression become the arguments of a structure (this is described in section 6.2.4).
   - In case of a structure as functional object, the structure arguments are inserted behind the place where the structure pointer is on the stack.
   - Finally, as in the `ll-call` instruction, the return address is pushed on the stack and execution is continued at @*function*/n or @*func*/*arity*, respectively. If the function is a builtin (e.g. `+`), the function/relation implementing it is simply executed and the result is put on the stack.
2. `apply` n: In case of `apply`, there is only a minor difference to `funcall`: the top stack element, which must be a list, is expanded on the stack, and the length of this list minus 1 is added to the arity of the function to be called.

3. `eval`: The external representation (denoted as $\widetilde{expr}$) of the expression which has to be evaluated is embedded in a new function definition `(defun %eval () ` $\widetilde{expr}$ `)` which is executed. Subsequently, the address of the instruction behind the `eval` instruction is pushed as return address on the stack and execution is continued with `%eval`.

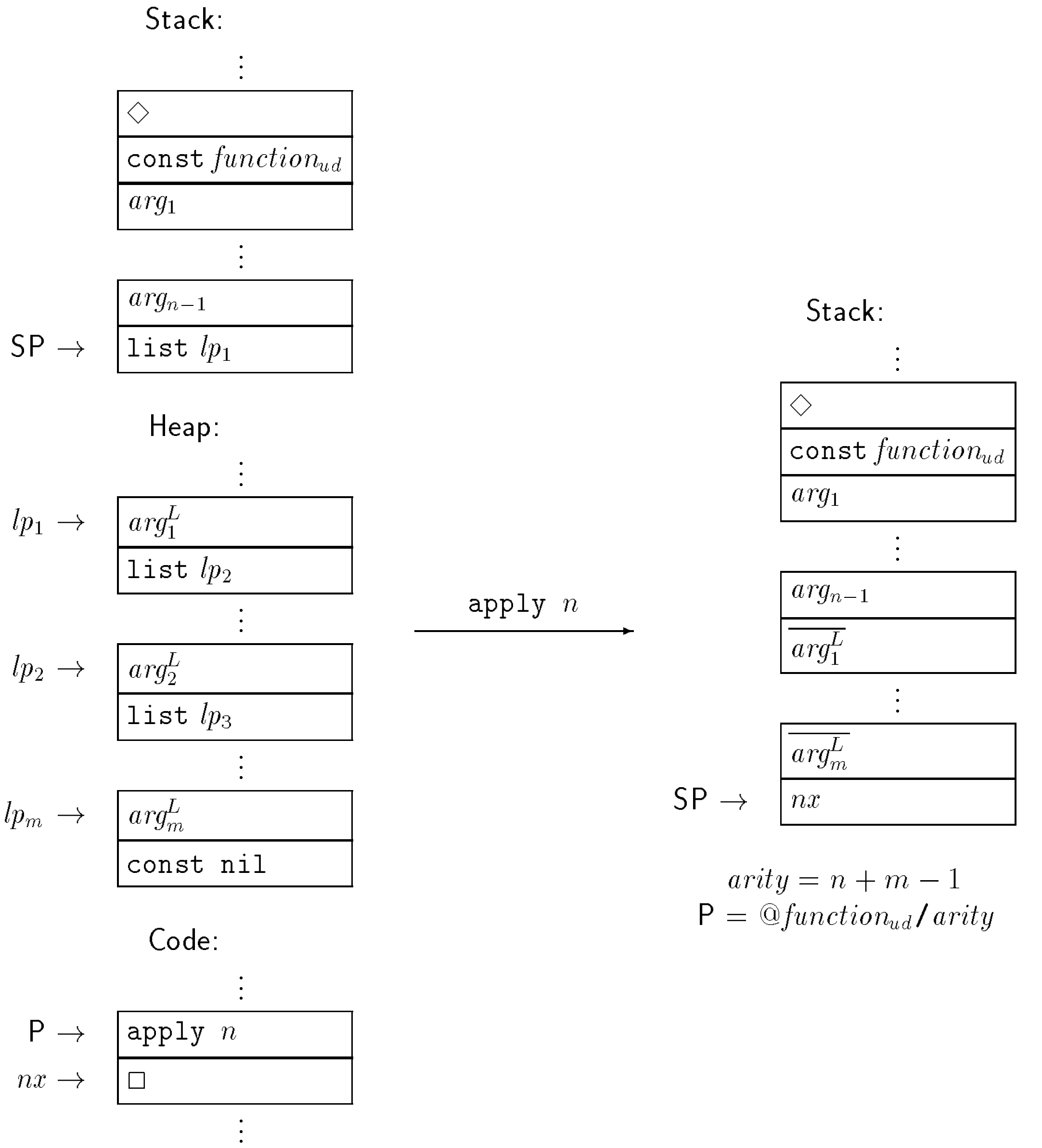
   Note that for `eval` the compiler is used at run time instead of *interpreting* the expression.





$func_{ud}/arity$ is user-defined

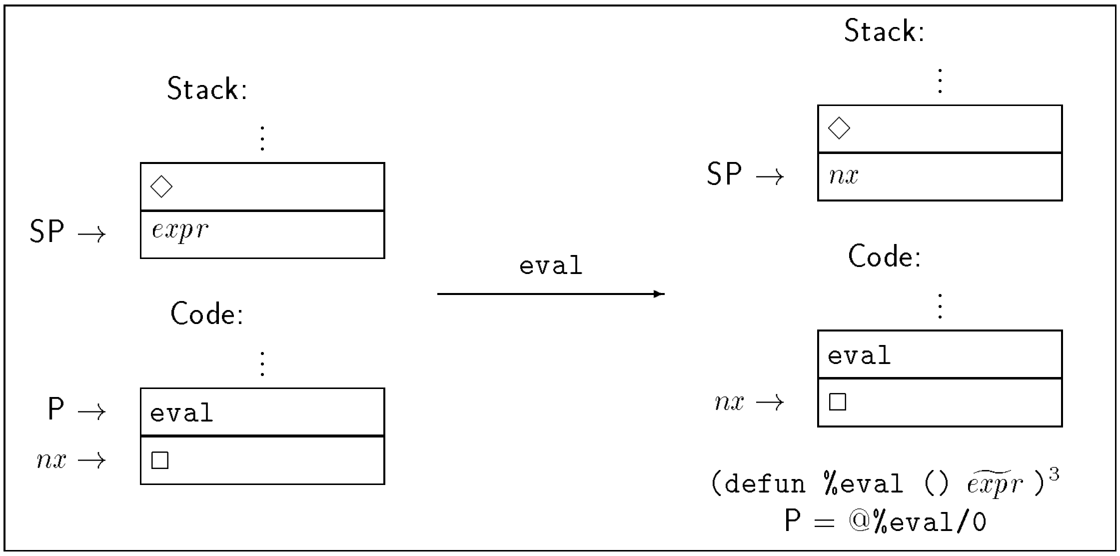
For builtins, the functions implementing these builtins are directly evaluated; their return value is put on the stack.



$function_{ud}/n$ is user defined

If the functional object is not a constant but a structure, this structure is expanded on the stack just as in `funcall`.

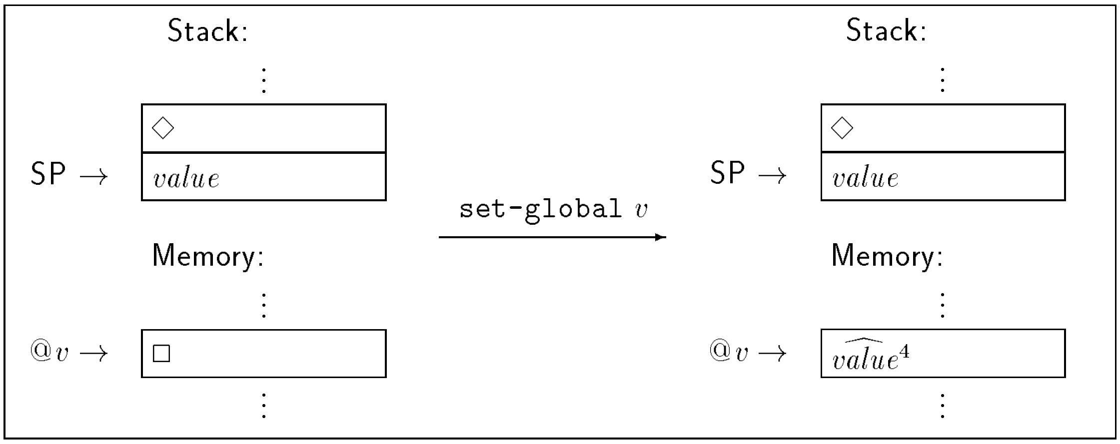
For builtins, the functions implementing these builtins are directly evaluated; their return value is put on the stack.



5.4.8 Global Variables

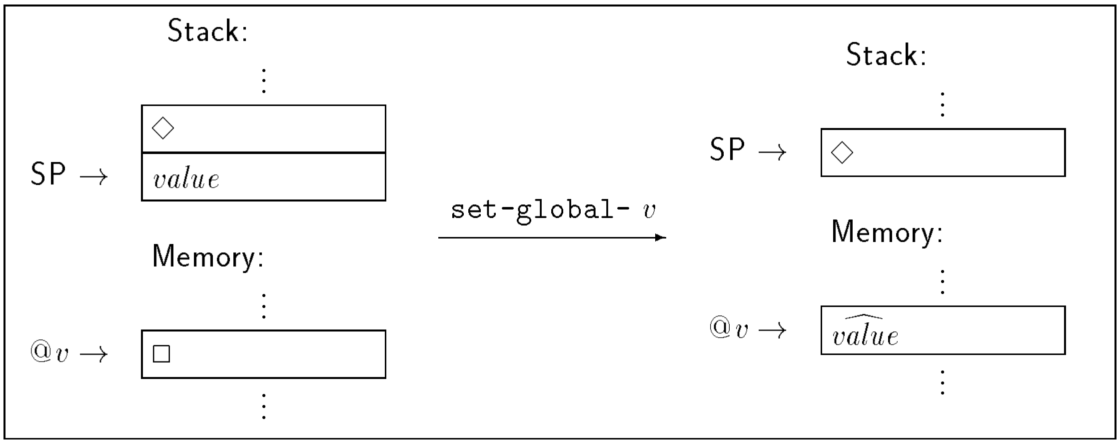
The LLAMA instruction for setting a global variable, which is used in the compilation of `setq` and `psetq`, is `set-global` *v*: the expression (which must not contain free logic variables) is copied into the general purpose memory (Memory), and its toplevel representation is stored in the address corresponding to *v*.

This copying is needed because the value of global variables should be stored permanently. Stack and heap are reset between queries, thus the general purpose memory is used.

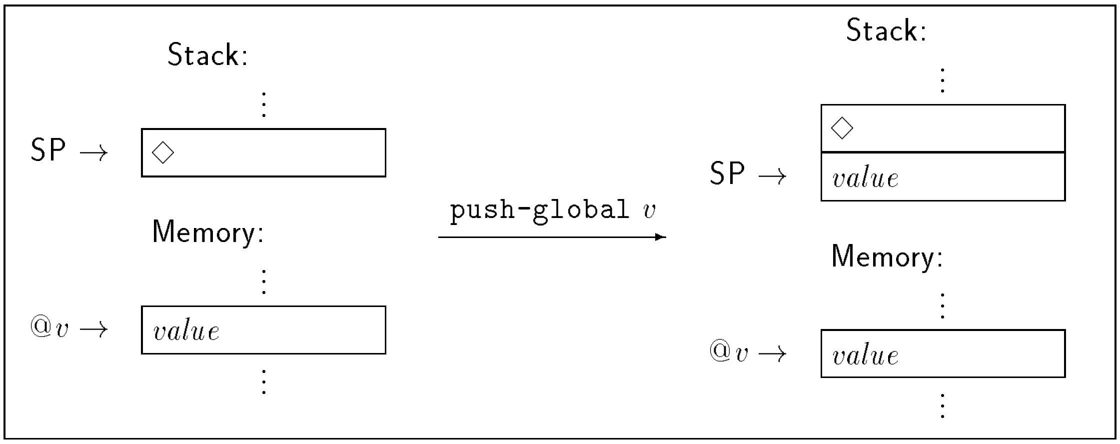


If the value which is stored in the global variable is not needed afterwards, it is removed in the `set-global-` instruction thus avoiding a subsequent `pop` instruction (see section 6.3).

[3] $\widetilde{expr}$ is the external representation of *expr*.
[4] $\widehat{value}$ is a copy of *value* in Memory.



The counterpart to the `dup` n instruction for local variables is `push-global` v in the case of global variables.

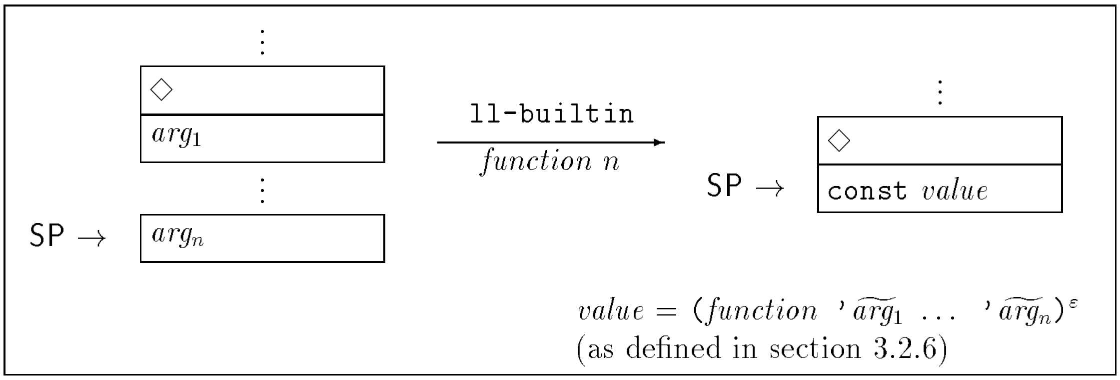


5.4.9 Numerical and String Builtins

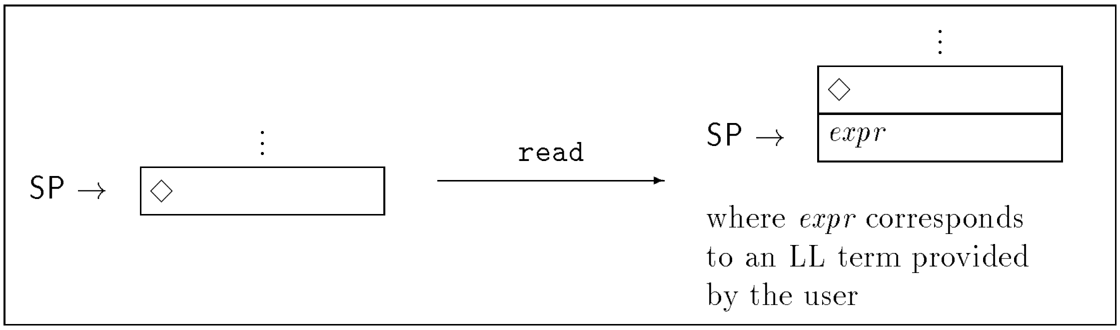
Builtins expecting (a usually variable number of) constants (numbers or strings) are handled by a single LLAMA instruction `ll-builtin`; this instruction was introduced to keep the number of different LLAMA instructions as small as possible.

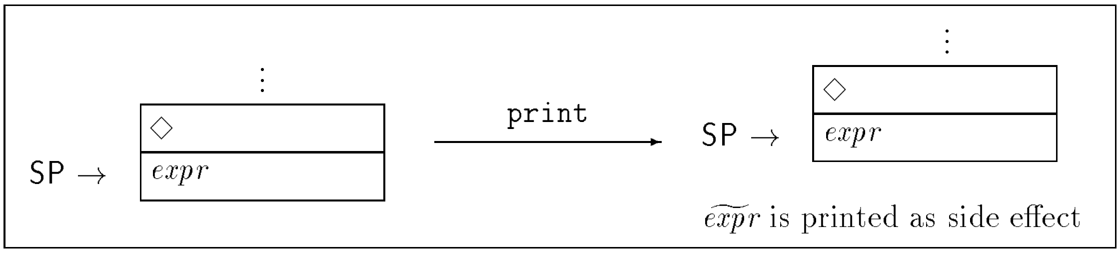
constant builtins
`+ - * / 1+ 1-`
`= /= < > <= >=`
`string< string>`

In our prototypical COMMON LISP implementation, all these builtins can be handled by COMMON LISP itself. If they should be re-implemented in any other language, the definitions given in section 3.2.6 can be used.



5.4.10 I/O Builtins





Chapter 6

Compiling LL into the LLAMA

The compiler $<>$ maps LL expressions into LLAMA instructions:

$$<>: \mathit{EXPR} \times \mathit{ENV} \rightarrow \mathit{INSTRSEQ}$$

with	*EXPR*	is the set of all LL expressions as described in chapter 3
	ENV	is the set of compiler environments used to keep track of the stack positions of local variables (see section 6.1)
	INSTRSEQ	is the set of LLAMA instruction sequences

For writing convenience, the function symbol $<>$ will only be used as in:

$$< \mathit{expr} >_{env} = \mathit{instrseq}$$

In this chapter, a precise description of the LL compiler via *transformation rules* is given (section 6.2). This allows a simple re-implementation in any (functional or imperative) programming language.

In these transformation rules, the compiler environment and the peep-hole optimizer are not included. They are described in section 6.1 and 6.3.

In section 6.4, some examples illustrating the transformation rules are presented.

6.1 The Compiler Environment

In the LLAMA, parameters passed to a function and local variables are accessed via the `dup` instruction on the stack. For that reason, the stack position of parameters and variables has to be known at compile time. This task is performed by the compiler environment: it contains pairs $\langle \mathit{var}\ \mathit{pos} \rangle$ associating parameters and local variables with stack positions.

Let us consider the following LL function:

```
(defun f5 (x y) (+ x 5 y))
```

The code produced by <> is[1]:

```
.proc f5/2
  dup 2              % variable x
  push-constant 5
  dup 3              % variable y
  ll-builtin + 3
  return 2
.end
```

The arguments for the two `dup` instructions were determined with the help of the environment. As shown in the following diagram, the compiler has to simulate the effects of the instructions on the stack in order to keep track of the parameter positions:

Instruction		*Stack*	*Environment*
`.proc f5/2`		$\boxed{x^{2}}\boxed{y^{1}}\boxed{retaddr^{0}}$	$\{\langle x,2\rangle,\langle y,1\rangle\}$
`dup 2`	$\xrightarrow{x}$	$\boxed{x^{3}}\boxed{y^{2}}\boxed{retaddr^{1}}\boxed{x^{0}}$	$\{\langle x,3\rangle,\langle y,2\rangle\}$
`push-constant 5`	$\xrightarrow{5}$	$\boxed{x^{4}}\boxed{y^{3}}\boxed{retaddr^{2}}\boxed{x^{1}}\boxed{5^{0}}$	$\{\langle x,4\rangle,\langle y,3\rangle\}$
`dup 3`	$\xrightarrow{y}$	$\boxed{x^{5}}\boxed{y^{4}}\boxed{retaddr^{3}}\boxed{x^{2}}\boxed{5^{1}}\boxed{y^{0}}$	$\{\langle x,5\rangle,\langle y,4\rangle\}$
`ll-builtin + 3`	$\longrightarrow$	$\boxed{x^{3}}\boxed{y^{2}}\boxed{retaddr^{1}}\boxed{x{+}5{+}y^{0}}$	$\{\langle x,3\rangle,\langle y,2\rangle\}$
`return 2`	$\longrightarrow$	$\boxed{x{+}5{+}y^{0}}$	$\{\}$

In the transformation rules given in section 6.2, these changes of the environment are not made explicit. Only if variables are *added* to the environment (as in `defun` and `let`), the environment extension is shown.

6.2 Transformation Rules

In the following sections, the compiler <> will be described with a system of transformation rules; these rules can be of the following forms:

[1] The assembler instructions `.proc` and `.end` mark the beginning and end of a function or relation.

$\boxed{\text{form } A}$	$\boxed{\text{form } B}$	$\boxed{\text{form } C}$
$\dfrac{\mathit{expression}}{\begin{array}{c} \mathit{instruction}_1 \\ \ldots \\ \mathit{instruction}_n \end{array}}$	$\dfrac{\mathit{expression}}{\begin{array}{c} \mathit{instruction}_1 \\ \ldots \\ < \mathit{expression}' > \\ \ldots \\ \mathit{instruction}_n \end{array}}$	$\dfrac{\mathit{expression}}{\begin{array}{c} \mathit{instruction}_1 \\ \ldots \\ < \mathit{expression}' >_{var_1 \ldots var_n} \\ \ldots \\ \mathit{instruction}_n \end{array}}$

(A) Rules of form A map an *expression* $\in$ *EXPR* directly into an instruction sequence $\langle \mathit{instruction}_1, \ldots, \mathit{instruction}_n \rangle \in \mathit{INSTRSEQ}$.

(B) In rules of form B, recursive calls to $<>$ are expressed by $< \mathit{expression}' >$ parts on the right-hand side of the rules.

(C) If recursive calls of $<>$ require the environment of local variables to be extended (as in `defun` and `let`), these variables are added as an index to $<>$ as in $< \mathit{expression}' >_{var_1 \ldots var_n}$.

6.2.1 `defun`

$$\frac{\texttt{(defun } \mathit{function}\ (var_1 \ldots var_n)\ \ .\ \ \mathit{body})}{\begin{array}{l} \texttt{.proc } \mathit{function/n} \\ < \texttt{(progn} \ \ .\ \ \mathit{body}) >_{var_1 \ldots var_n} \\ \texttt{return } n \\ \texttt{.end} \end{array}}$$

The assembler statements `.proc` and `.end` do not generate any code but are used to mark the beginning and end of a relation or function (see chapter 7).

6.2.2 Simple Expressions, Lists, and Structures

$$\frac{\mathit{constant}}{\texttt{push-constant } \mathit{constant}}$$

where *constant* is a number, string, `nil`, `t`

$$\frac{\mathit{local\text{-}variable}}{\texttt{dup } \mathit{stack\text{-}position}}$$

where *stack-position* is the position of *local-variable* on the stack

$$\frac{\textit{global-variable}}{\texttt{push-global}\ \textit{global-variable}}$$

$$\frac{\texttt{[}\textit{functor}\ \textit{arg}_1\ \ldots\ \textit{arg}_n\texttt{]}}{<\ \texttt{(struct}\ \texttt{'}\textit{functor}\ \texttt{'}\textit{arg}_1\ \ldots\ \texttt{'}\textit{arg}_n\texttt{)}\ >}$$

$$\frac{\texttt{'}\textit{atom}}{\texttt{push-constant}\ \textit{atom}}$$

$$\frac{\texttt{'[}\textit{functor}\ \textit{arg}_1\ \ldots\ \textit{arg}_n\texttt{]}}{<\ \texttt{[}\textit{functor}\ \textit{arg}_1\ \ldots\ \textit{arg}_n\texttt{]}\ >}$$

$$\frac{\texttt{'(}\textit{arg}_1\ \ldots\ \textit{arg}_n\texttt{)}}{<\ \texttt{(list}\ \texttt{'}\textit{arg}_1\ \ldots\ \texttt{'}\textit{arg}_n\ \texttt{)}\ >}$$

$$\frac{\texttt{(list}\ \textit{arg}_1\ \ldots\ \textit{arg}_n\texttt{)}}{<\ \texttt{(cons}\ \textit{arg}_1\ \texttt{(cons}\ \textit{arg}_2\ \ldots\texttt{(cons}\ \textit{arg}_n\ \texttt{nil)}\ldots\texttt{)}\ >}$$

$$\frac{\texttt{(struct}\ \textit{functor}\ \textit{arg}_1\ \ldots\ \textit{arg}_n\texttt{)}}{\begin{array}{l} <\ \textit{functor}\ > \\ <\ \textit{arg}_1\ > \\ \ldots \\ <\ \textit{arg}_n\ > \\ \texttt{struct}\ n \end{array}}$$

6.2.3 `setq` and Relatives

$$\frac{\texttt{(setq}\ \textit{var}_1\ \textit{expr}_1\ \ldots\ \textit{var}_n\ \textit{expr}_n\texttt{)}}{<\ \texttt{(progn (\%setq}\ \textit{var}_1\ \textit{expr}_1\texttt{)}\ \ldots\ \texttt{(\%setq}\ \textit{var}_n\ \textit{expr}_n\texttt{))}\ >}$$

$$\frac{\texttt{(\%setq}\ \textit{local-variable}\ \textit{expression}\texttt{)}}{\begin{array}{l} <\ \textit{expression}\ > \\ \texttt{set-nth}\ \textit{stack-position} \end{array}}$$

where *stack-position* is the position of *local-variable* on the stack

$$\frac{\texttt{(\%setq}\ \textit{global-variable}\ \textit{expression}\texttt{)}}{\begin{array}{l} <\ \textit{expression}\ > \\ \texttt{set-global}\ \textit{global-variable} \end{array}}$$

`(psetq` var_1 $expr_1$ … var_n $expr_n$`)`

$< \mathrm{expr}_1 >$
…
$< \mathrm{expr}_n >$
destr. set statement for var_n
…
destr. set statement for var_1
`push-constant nil`

where the destructive set statement for var_i is `set-nth-` *stack-position*$_i$ for local and `set-global-` var_i for global variables (cf. `%setq`)

6.2.4 `funcall` and Relatives

`(funcall` *function* $expr_1$ … $expr_n$`)`

$<$ *function* $>$
$< \mathrm{expr}_1 >$
…
$< \mathrm{expr}_n >$
`funcall` n
`remove 1`

`remove 1` removes the functional object, the result of $<$ *function* $>$, from the stack

`(apply` *function* $expr_1$ … $expr_n$`)`

$<$ *function* $>$
$< \mathrm{expr}_1 >$
…
$< \mathrm{expr}_n >$
`apply` n
`remove 1`

for `remove 1`, see `funcall`;
$expr_n$ must evaluate to a list

`(function` *symbol*`)`

$<$ '*symbol* $>$

`(function (lambda (`v_1 … v_n`) .` *body*`))`

$<$ '*symbol* $>$

with:

- *symbol* is a new symbol
- *body* does not contain any free local variables
- `(defun` *symbol* `(`v_1… v_n`)` `.` *body*`)` is executed

$$\begin{array}{l} \texttt{(function (lambda (}v_1 \ \ldots v_n\texttt{) . }\mathit{body}\texttt{))} \\ \hline \qquad < \texttt{(struct '}\mathit{symbol}\ f_1 \ \ldots f_m\texttt{)} > \end{array}$$

with:

- *symbol* is a new symbol
- $f_1 \ldots f_m$ are the free local variables of *body*
- `(defun` *symbol* `(`$f_1 \ldots f_m\ v_1 \ldots v_n$`)` `.` *body*`)` is executed

$$\begin{array}{l} \texttt{(eval }\mathit{expression}\texttt{)} \\ \hline \quad < \mathit{expression} > \\ \quad \texttt{eval} \end{array}$$

6.2.5 `let` and `let*`

$$\begin{array}{l} \texttt{(let ((}v_1\ \ e_1\texttt{)}\ \ldots\texttt{(}v_n\ \ e_n\texttt{))}\ \texttt{.}\ \mathit{body}\texttt{)} \\ \hline \qquad < e_1 > \\ \qquad \ldots \\ \qquad < e_n > \\ \qquad < \texttt{(progn . }\mathit{body}\texttt{)} >_{v_1 \ldots v_n} \\ \qquad \texttt{remove}\ n \end{array}$$

$$\begin{array}{c} \texttt{(let* ((}v_1\ \ e_1\texttt{)}\ \ldots\texttt{(}v_n\ \ e_n\texttt{))}\ \texttt{.}\ \mathit{body}\texttt{)} \\ \hline < \texttt{(let ((}v_1\ \ e_1\texttt{)) (let ((}v_2\ \ e_2\texttt{))}\ \ldots\texttt{(let ((}v_n\ \ e_n\texttt{))}\ \texttt{.}\ \mathit{body}\texttt{))}\ldots\texttt{)} > \end{array}$$

6.2.6 `if` and Relatives

$$\begin{array}{l} \texttt{(if }\mathit{test}\ \ \mathit{expr}_1\ \ \mathit{expr}_2\texttt{)} \\ \hline \quad < \mathit{test} > \\ \quad \texttt{on-nil-goto}\ \mathit{label}_A \\ \quad < \mathit{expr}_1 > \\ \quad \texttt{goto}\ \mathit{label}_B \\ \mathit{label}_A\texttt{:} \\ \quad < \mathit{expr}_2 > \\ \mathit{label}_B\texttt{:} \end{array}$$

where $label_A$ and $label_B$ are new labels

$$\begin{array}{c} \texttt{(if }\mathit{test}\ \ \mathit{expression}\texttt{)} \\ \hline < \texttt{(if }\mathit{test}\ \ \mathit{expression}\ \texttt{nil)} > \end{array}$$

$$\frac{\texttt{(cond (}t_1\ \texttt{.}\ e_1\texttt{)}\ \ldots\texttt{(}t_n\ \texttt{.}\ e_n\texttt{))}}{\begin{array}{l}<\texttt{(if}\ t_1\ \texttt{(progn . }e_1\texttt{)}\\ \qquad\qquad \texttt{(if}\ t_2\ \texttt{(progn . }e_2\texttt{)}\ \ldots\texttt{(if}\ t_n\ \texttt{(progn . }e_n\texttt{) nil)}\ldots\texttt{)}>\end{array}}$$

$$\frac{\texttt{(and)}}{<\texttt{t}>}$$

$$\frac{\texttt{(and}\ expression\texttt{)}}{<expression>}$$

$$\frac{\texttt{(and}\ expr_1\ \ldots expr_n\texttt{)},\ n \geq 2}{<\texttt{(if}\ expr_1\ \texttt{(if}\ expr_2\ \ldots\texttt{(if}\ expr_{n-1}\ expr_n\texttt{)}\ldots\texttt{)}>}$$

$$\frac{\texttt{(or)}}{<\texttt{nil}>}$$

$$\frac{\texttt{(or}\ expression\texttt{)}}{<expression>}$$

$$\frac{\texttt{(or}\ expr_1\ \ldots expr_n\texttt{)},\ n \geq 2}{<\texttt{(\%if}\ expr_1\ \texttt{(\%if}\ expr_2\ \ldots\texttt{(\%if}\ expr_{n-1}\ expr_n\texttt{)}\ldots\texttt{)}>}$$

$$\frac{\texttt{(\%if}\ expr_1\ expr_2\texttt{)}}{<\texttt{(let ((}var\ expr_1\texttt{)) (if}\ var\ var\ expr_2\texttt{))}>}$$ where *var* is a new variable

6.2.7 Sequential Evaluation

$$\frac{\texttt{(progn}\ expr_1\ \ldots\ expr_n\texttt{)}}{\begin{array}{l}<expr_1>\\ \texttt{pop}\\ \ldots\\ <expr_{n-1}>\\ \texttt{pop}\\ <expr_n>\end{array}}$$

6.2.8 Non-local Exits and Loops

`(catch` *tag* `.` *body*`)`

< *tag* >
`catch` *label* — where *label* is a new label
< `(progn . ` *body*`)` > — (for `throw`, see section 6.2.9)
`remove-tag`
label:

`(loop .` *body*`)`

`catch-nil` $label_B$
$label_A$:
< `(progn .` *body*`)` > — where $label_A$ and $label_B$ are new labels
`pop`
`goto` $label_A$
$label_B$:

`(return` *expression*`)`

< `(throw-nil` *expression*`)` > — (for `throw-nil`, see section 6.2.9)

`(do` *init* *exit* `.` *body*`)`,
$init$ = `((`var^I_1 $expr^I_{1_a}$ $expr^I_{1_b}$`)`...`(`var^I_n $expr^I_{n_a}$ $expr^I_{n_b}$`))`,
$exit$ = `(`$test$ $expr^E_1 \dots expr^E_m$`)`

< `(let ((`var^I_1 $expr^I_{1_a}$`)`...`(`var^I_n $expr^I_{n_a}$`))`
    `(loop`
     `(if` *test*
        `(progn` $expr^E_1 \dots expr^E_{m-1}$ `(return` $expr^E_m$`))`
        `(progn ,@`*body* `(psetq (`var^I_1 $expr^I_{1_b}$`)`...`(`var^I_n $expr^I_{n_b}$`)))))` >[2]

6.2.9 Simple Builtins

`(`$function_{va}$ $expr_1 \dots expr_n$`)`

< $expr_1$ > — where $function_{va}$ is one of
...
< $expr_n$ >
`ll-builtin` $function_{va}$ n

```
+ - * / 1+ 1-
= /= < > <= >=
string< string>
```

---

[2] `,@`*list* splices *list* into the embedding list.

$$\frac{(\mathit{function}_{fa}\ \ \mathit{expr}_1 \ldots \mathit{expr}_n)}{\begin{array}{l} < \mathit{expr}_1 > \\ \ldots \\ < \mathit{expr}_n > \\ \mathit{function}_{fa} \end{array}}$$

where $\mathit{function}_{fa}$ is one of

$\mathit{function}_{fa}$	n	$\mathit{function}_{fa}$	n
`cons`	2	`eq`	2
`car`	1	`eql`	2
`cdr`	1	`equal`	2
`null`	1		
`consp`	1	`print`	1
`functor`	1	`read`	0
`arity`	1		
`elt`	2	`throw`	2
`structp`	1	`throw-nil`	1

6.2.10 User-defined Functions

$$\frac{(\mathit{function}_{ud}\ \ \mathit{expr}_1 \ldots \mathit{expr}_n)}{\begin{array}{l} < \mathit{expr}_1 > \\ \ldots \\ < \mathit{expr}_n > \\ \texttt{ll-call}\ \mathit{function}_{ud}/n \end{array}}$$

6.3 The Peep-Hole Optimizer

The LL compiler sometimes produces code which by *local inspection* can be considerably improved. For these cases, a peep-hole optimizer is used which looks at the code and changes it to make it smaller and faster.

The most obvious inefficient constellations are those where something is first pushed on the stack and then directly discarded. The instructions responsible for this can simply be removed:

$$\frac{\begin{array}{l} \texttt{push-constant}\ \mathit{constant} \\ \texttt{pop} \end{array}}{\text{—}}$$

Another simplification is to collect `remove` instructions:

$$\frac{\begin{array}{l} \texttt{remove}\ n \\ \texttt{remove}\ m \end{array}}{\texttt{remove}\ n{+}m}$$

With this rule, a sequence of `remove` instructions is replaced by a single instruction. This situation occurs when multiple `let` forms end at a single location.

In the LLAMA, some instructions were introduced especially for the peep-hole optimizer: `set-nth-` and `set-global-`. Their counterparts without a trailing dash leave the top stack element on the stack, the versions with the dash remove it. An instruction sequence consisting of `set-nth` or `set-global` and a `pop` instruction (as it is produced by multiple `setq` statements) is replaced by the corresponding `set-nth-` or `set-global-` instruction:

```
 set-nth n
 pop
------------
set-nth- n
```

```
 set-global n
 pop
---------------
set-global- n
```

Many additional peep-hole optimizations are possible, most of them also requiring new LLAMA instructions, e.g. `cons*` n (executing `cons` n times):

```
 cons
 cons
---------
cons* 2
```

```
 cons* n
 cons
-----------
cons* n+1
```

```
 cons
 cons* n
-----------
cons* n+1
```

6.4 Examples

In this section, some examples illustrating the compiler are presented.

Example 6.1
The first example, `fac`, shows the usage of `on-nil-goto` in the compilation of `if`:

```
(defun fac (x)
  (if (equal 0 x)
      1
      (* x (fac (- x 1)))))

.proc fac/1
  push-constant 0    % 0
  dup 2              % x
  equal              % (equal 0 x) ?
  on-nil-goto "L1"   % no, go to "L1"
  push-constant 1
  goto "L2"
"L1"
  dup 1              % x
  dup 2              % x
  push-constant 1    % 1
  ll-builtin - 2     % (- x 1)
  ll-call fac/1      % (fac (- x 1))
  ll-builtin * 2     % (* x (fac (- x 1)))
"L2"
  return 1
.end
```

Example 6.2
The second example shows how lists are accessed via `cons`, `car`, `cdr`, and `null`:

```
(defun append (l1 l2)
  (if (null l1)
      l2
      (cons (car l1) (append (cdr l1) l2))))

.proc append/2
  dup 2              % l1
  null               % (null l1) ?
  on-nil-goto "L1"   % no, go to "L1"
  dup 1              % l2
  goto "L2"
"L1"
  dup 2              % l1
  car                % (car l1)
  dup 3              % l1
  cdr                % (cdr l1)
  dup 3              % l2
```

```
  ll-call append/2   % (append (cdr l1) l2)
  cons               % (cons (car l1) (append (cdr l1) l2))
"L2"
  return 2
.end
```

Example 6.3
This example shows how higher-order functions are compiled:

```
(defun mapcar (f l)
  (if l
      (cons (funcall f (car l))
            (mapcar f (cdr l)))))

.proc mapcar/2
  dup 1              % l
  on-nil-goto "L1"
  dup 2              % f
  dup 2              % l
  car                % (car l)
  funcall 1          % (funcall f (car l))
  remove 1           % remove f from stack
  dup 3              % f
  dup 3              % l
  cdr                % (cdr l)
  ll-call mapcar/2   % (mapcar f (cdr l))
  cons               % (cons (funcall f (car l)) (mapcar f (cdr l)))
  goto "L2"
"L1"
  push-constant nil
"L2"
  return 2
.end
```

For lambda expressions containing free variables, structures as functional objects are constructed:

```
(defun listadd (l n) ; add n to each element in l
  (mapcar #'(lambda (x) (+ x n)) l))

.proc listadd/2
  push-constant lambda43
  dup 2              % n
  struct 1           % [lambda43 n]
```

```
  dup 3              % l
  ll-call mapcar/2   % (mapcar l [lambda43 n])
  return 2
.end

.proc lambda43/2     % (defun lambda43 (n x) (+ x n))
  dup 1              % x
  dup 3              % n
  ll-builtin + 2     % (+ x n)
  return 2
.end
```

Example 6.4
This example illustrates how loops are compiled into `catch-nil` and `throw-nil`:

```
(defun reverse (list)
  (do ((l list (cdr l))
       (res nil (cons (car l) res)))
      ((null l) res)))

.proc reverse/1
  dup 1              % list   (initialization
  push-constant nil  % nil     for l and res)
  catch-nil "L4"
"L1"
  dup 1              % l
  null               % (null l) ?
  on-nil-goto "L2"
  dup 0              % res
  throw-nil          % leave loop with res as return value
"L2"
  dup 1              % l
  cdr                % (cdr l)
  dup 2              % l
  car                % (car l)
  dup 2              % res
  cons               % (cons (car l) res)
  set-nth- 2         % res <- (cons (car l) res))
  set-nth- 2         % l   <- (cdr l)
  goto "L1"
"L4"
  remove 2           % remove l and res
  return 1
.end
```

Chapter 7

Integrating Abstract Machines: The GAMA

GAMA, the General Abstract Machine Assembler, is a programming environment supporting the development and integration of abstract machines. In this work, it was used to integrate an existing implementation of the WAM (our version of the NyWAM [Nystrøm, 1985]) with the newly developed LLAMA (chapter 5).

In the following sections, 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.

7.1 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[1]:

- `(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

[1] 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).

- `(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*

7.2 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).

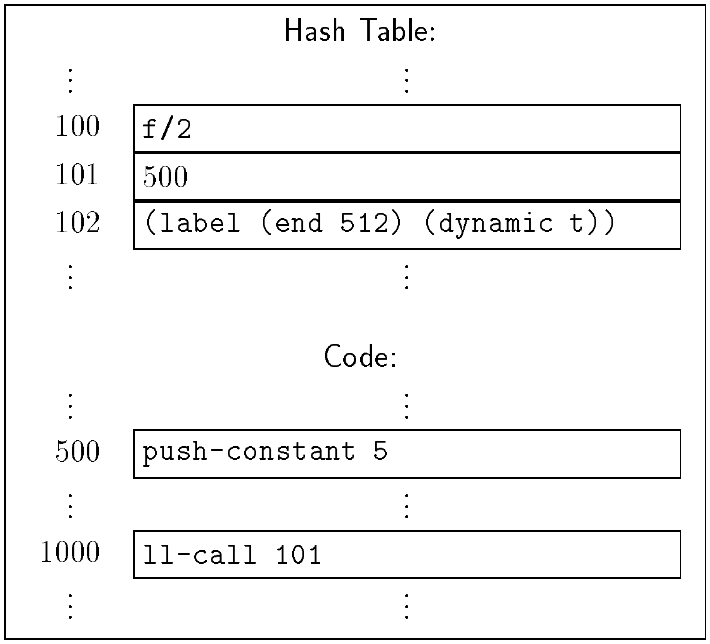
The following functions are defined on hash tables:

- `(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 `ll-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 `ll-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.

7.3 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[2], and the COMMON LISP code defining the instruction.

The following example shows the definition of the LLAMA instruction `push-constant`:

```
(definstr push-constant (c) (CONST)
  :standard (ll.push-constant
             (stack-push (constant c))))
```

`ll.push-constant` is the name of the COMMON LISP function corresponding to the `push-constant` instruction. The keyword `:standard` declares

[2]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 `ll-builtin`), `LABEL` for labels, `VARIABLE` for global (LL) variables, `HASHTABLE` for hash tables (used in the WAM switch instructions), and `X` for arbitrary arguments. Additional types can be defined with `gasm.deftype`.

`push-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 ll-call (proc) (LABEL)
  :static  (ll.call/st
            (stack-push (reg P))
            (set-reg P proc))
  :dynamic (ll.call/dy
            (stack-push (reg P))
            (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).

7.4 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 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 7.3)
- `.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 7.1
The assembler and machine code (with the corresponding hash table entry) for the LL function

```
(defun fac (arg#1)
  (if (equal 0 arg#1) 1 (* arg#1 (fac (- arg#1 1)))))
```

is as follows:

Assembler code	Hash table entry and machine code
`.module lluser`	Hash Table (*for module* `lluser`):
`.proc`	`262976:  fac/1`
`.dynamic`	`262977:  263862`
`fac/1`	`262978:  (label (end 263874) (dynamic t))`
	Code:
`    push-constant 0`	`263862:  push-constant 0`
`    dup 2`	`263863:  dup 2`
`    equal`	`263864:  equal`
`    on-nil-goto "l1"`	`263865:  on-nil-goto 263868`
`    push-constant 1`	`263866:  push-constant 1`
`    goto "l2"`	`263867:  goto 263874`
`"l1"`	
`    dup 1`	`263868:  dup 1`
`    dup 2`	`263869:  dup 2`
`    push-constant 1`	`263870:  push-constant 1`
`    ll-builtin - 2`	`263871:  ll-builtin - 2`
`    ll-call fac/1`	`263872:  ll-call 262977`
`    ll-builtin * 2`	`263873:  ll-builtin * 2`
`"l2"`	
`    return 1`	`263874:  return 1`
`.end`	

△

Chapter 8

Conclusions and Future Work

In this work, a general and flexible technology for loosely integrating relational and functional languages on the basis of abstract machines was developed.

We first highlight several aspects of this integration before discussing its achievements and possible enhancements:

- Relational and functional languages, when individually compilable into abstract machines (as in the case of PROLOG-like and LISP-like languages), can be integrated by extending their abstract machines by only a few, usually very simple instructions.
- The mutual access thus achieved on the abstract machine level is then used to integrate the languages on the source level. In the case of PROLOG-like relational languages (e.g. REL), the `is` builtin is generalized to allow functions defined in the functional language in addition to (usually builtin) arithmetical functions on its right-hand side. The functional language, if desired, can access relations either via `once`, retrieving only the first solution, or a `bagof`-like construct, retrieving all solutions.
- Deterministic predicates, for which a PROLOG computation rule foundation is developed in this work, can be detected and transformed into functions, thus improving their efficiency considerably since the overhead of unification and backtracking is avoided.

Let us mention here three major contributions of our functional-plus-logic programming on an integrated platform (FLIP):

- FLIP can be regarded as an implementation technique for relational languages (e.g. PROLOG) and relational-functional languages (e.g. RELFUN), where deterministic predicates exploit special-purpose compilation into a functional stack machine (LLAMA) loosely coupled to the language's principal abstract machine. In the case of RELFUN, the functional language

LL needed for this implementation can even be regarded as a version of the functional source language component itself.

- Algorithms from existing software libraries can be combined even if specified in different paradigms without the need of re-implementation. Since the development and maintenance of software is very expensive, the loss of some declarativeness usually caused by hybrid integrations is justifiable.

- Since the seminal paper [Warren *et al.*, 1977] there has been a dispute about the relative efficiencies of LISP and PROLOG: the authors gave examples where their PROLOG was as efficient as the LISPs of that time.

  In the current work we contribute to the efficiency of both language paradigms individually and in combination. While it is difficult to assert a definitive efficiency advantage for one of these paradigms, our benchmarks tend to support the view that LISP can still be made faster than PROLOG: in case of the integration of REL and LL, a factor of about 2-4 was achieved by transforming deterministic predicates into LL functions. This is mainly caused by LL being on a lower (though still declarative) level than relational languages, i.e. allowing much more operational specifications.

In the future, this integration can be improved in several directions:

- In the current implementation, the transformation of deterministic predicates into LL functions does not contain optimizations exploiting the extra-functional builtins of LL. In the future, optimizations such as transforming tail-recursions into loops or using `case` statements[1] in addition to the `if`/`let` cascade (analogously to PROLOG indexing; see section 4.2.3.4) can be developed to further improve the efficiency.

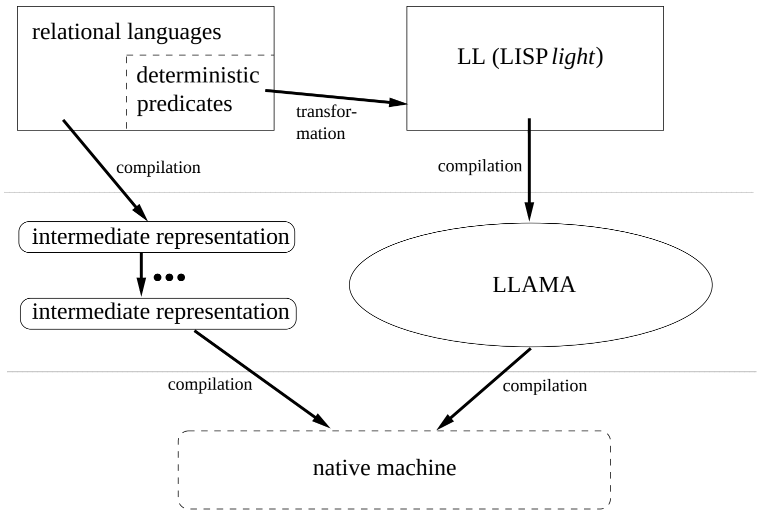
- As was laid out in section 4.2.3.4, predicates automatically detected and transformed into deterministic functions have to contain quasi-final cuts and catch-all clauses. A remedy to avoid these is the introduction of type definitions and signatures (see example 4.7 on page 39), as they are currently developed for RELFUN [Hall, 1995]. New algorithms, detecting determinism in the more declarative context of mode declarations together with signatures, have to be conceived: not only modes, but also types should be handled by abstract interpretation.

- Since the structure of the LLAMA is very similar to native machines, its instructions — unlike WAM instructions — can easily be compiled into native machine code. In future research it could be examined how LL

[1] `case` is not yet implemented.

and the LLAMA can be useful in the compilation of PROLOG and other relational languages into native code [Taylor, 1990; Van Roy, 1994].

A possible scenario for this is illustrated in the following diagram:



- The loose integration, performed on the abstract machine level, is not restricted to relational and functional languages. Some research could be done to identify additional programming paradigms integratable in an analogous manner. These programming paradigms could either be declarative, like constraint logic or object-oriented programming, or rather operational, in order to access programming languages whose software libraries can be reused.

Appendix A

Benchmarks

In our prototypical implementation, WAM and LLAMA code are interpreted by a COMMON LISP program. It would thus be unfair to compare the run time behaviour of LL programs with PROLOG programs compiled into C interpreted WAM code.

A.1 Naive Reverse

The standard benchmark, naive reverse, was tested in three environments:

1. in REL, compiled into the WAM without indexing[1]
2. in REL with indexing
3. in LL, automatically generated from the REL specification (see section 4.2.3.5)

For naive reverse applied to a list with 50 elements, the following internal run times were measured:

1.	REL without indexing	0.596 s
2.	REL with indexing	0.295 s
3.	LL (automatically generated)	0.142 s

The speed up achieved by indexing and by transformation into LL (compared with the indexed version) are both approximately 2.

[1] For a survey of indexing, see [Stein, 1993] and [Sintek, 1993]

A.2 Naive Fibonacci

For the (naive) Fibonacci algorithm applied to 20, the following internal run times were measured:

1.	REL without indexing	11.1 s
2.	REL with indexing	6.23 s
3.	LL (automatically generated)	1.76 s

This time, the speed up achieved by indexing is 1.78, and the the speed up achieved by transformation into LL is 3.54 (compared with the indexed version).

A.3 Quicksort

For quicksort applied to a list with 200 elements, the following internal run times were measured:

1.	REL without indexing	1.65 s
2.	REL with indexing	1.46 s
3.	LL (automatically generated)	0.76 s

The speed up achieved by indexing is only 1.13, whereas the the speed up achieved by transformation into LL is 1.92 (compared with the indexed version).

Appendix B

The User Interface

In this appendix, the user interface for the prototypical implementation of the integration of REL, LL, WAM, and LLAMA in the RELFUN environment is described.

B.1 RELFUN Toplevel

In FLIP-extended RELFUN, three different toplevels are present: the interpreter, the emulator, and the LL toplevel. They are entered with `inter`, `emul`, and `ll`, respectively.

For the commands available at the interpreter and emulator toplevels, refer to the RELFUN documentation [Boley *et al.*, 1991] and [Boley *et al.*, 1993].

At the LL toplevel, the following commands are available:

- `load` *filename*: load and compile an LL file (if no extension is present, "`.lisp`" is assumed)
- `spy`: switch to spy mode (the executed machine code is displayed)
- `nospy`: finish spy mode
- `asm mod lluser`: show contents of LL name space
- `asm l` a b: display a memory area (addresses $a - b$)

B.2 Declarations

In RELFUN, declarations are realized via `declare` facts (in contrast to PROLOG, where declarations are usually goals, i.e. they are preceded by "`:-`").

LL functions, which should be accessed from REL, have to be declared:

- `declare( ll[`$function_1$`,...,`$function_n$`]).`
  declares ordinary functions to be accessible

- `declare( llp[`$testpred_1$`,...,`$testpred_n$`]).`
  declares the $testpred_i$ to be functions used as test predicates (i.e. a returned `nil` generates a failure)

Mode declarations have the following form:

`declare( mode[`$predicate$`[`$\mu_1, \ldots, \mu_n$`]] ).`

where the μ_i are either `g` or `x` (corresponding to $\mathcal{G}$ and $\mathcal{X}$ in section 4.2.2).

For example, the mode declarations for `append` and `reverse` are

```
declare(mode[append[g,g,x]]).
```

and

```
declare(mode[reverse[g,x]]).
```

If a predicate should additionally be declared to be representable as a (total) deterministic function (as was mentioned in subsection 4.2.3.5), `mode` is replaced by `dfmode`: `declare(dfmode[append[g,g,x]]).`

In appendix C, the usage of mode declarations is illustrated.

B.3 Transforming Deterministic REL Predicates into LL Functions

After loading REL programs with `consult` *filename* at the emulator toplevel, or asserting REL clauses (and mode declarations) with `az` *clause*, the following commands have to be used to detect and transform deterministic predicates and to compile the remaining (non-deterministic) predicates:

1. `undeclare` "executes" the `declare` facts and thus the mode declarations

2. `deta` transforms deterministic predicates into LL functions and compiles them

3. `compile` compiles the remaining (non-deterministic) predicates

These steps also are illustrated in appendix C.

Appendix C

Sample Dialog

```
rfe-p> destroy
rfe-p>
rfe-p> az declare( mode[ fac[g,x] ]).
rfe-p>
rfe-p> az fac(0,1) :- !.
rfe-p> az fac(X,Y) :- X1 is -(X,1), fac(X1, FX1), Y is *(X, FX1).
rfe-p>
rfe-p> az declare( mode[ f[g,x] ]).
rfe-p>
rfe-p> az f(s[X,Y], u[s[X,Y],s[A,B]]) :-
           A is -(B,A),
           s[A,B] is s[X,Y],
           fac(X,A),
           fac(Y,B) !.
rfe-p> az f(X, [X,X]).
rfe-p>
rfe-p> az tripfac(X, [X,Y]) :- fac(X,Y).
rfe-p> az tripfac(X, [X1,Y]) :- X1 is +(X,1), fac(X1,Y).
rfe-p> az tripfac(X, [X2,Y]) :- X2 is +(X,2), fac(X2,Y).
rfe-p>
rfe-p> az declare( mode[ tripfac1[g,x,x,x] ]).
rfe-p>
rfe-p> az tripfac1(X,F,F1,F2) :-
        fac(X,F),
        X1 is +(X,1), F1 is *(X1,F),
        X2 is +(X,2), F2 is *(X2,F1).
rfe-p>
rfe-p> undeclare
rfe-p> deta
rfe-p>
rfe-p> listing
tripfac(X, [X, Y]) :- Y is fac/2-1(X).
```

```
tripfac(X, [X1, Y]) :- X1 is +(X, 1), Y is fac/2-1(X1).
tripfac(X, [X2, Y]) :- X2 is +(X, 2), Y is fac/2-1(X2).
tripfac1(X1, X2, X3, X4) :- values[X2, X3, X4] is tripfac1/4-3(X1).
f(X1, X2) :- X2 is f/2-1(X1).
fac(X1, X2) :- X2 is fac/2-1(X1).

(defun tripfac1/4-3 (arg#1)
  (let ((v2082 (fac/2-1 arg#1)))
    (let ((v2083 (* (+ arg#1 1) v2082)))
      (struct 'values v2082 v2083 (* (+ arg#1 2) v2083)))))
tripfac1/4-3(X) :-
         F is fac/2-1(X),
         X1 is +(X, 1),
         F1 is *(X1, F),
         X2 is +(X, 2),
         F2 is *(X2, F1) &
         values[F, F1, F2].

(defun f/2-1 (arg#1)
  (if (and (structp arg#1) (equal 's (functor arg#1))
           (equal 2 (arity arg#1)))
      (let ((v2124 (elt arg#1 0)))
        (if (equal v2124 (fac/2-1 v2124))
            (let ((v2125 (elt arg#1 1)))
              (if (and (equal v2124 (- v2125 v2124))
                       (equal v2125 (fac/2-1 v2125)))
                  (struct 'u arg#1 arg#1)
                  (cons arg#1 (cons arg#1 nil))))
            (cons arg#1 (cons arg#1 nil))))
      (cons arg#1 (cons arg#1 nil))))
f/2-1(s[X, Y]) :-
         A is -(B, A),
         s[A, B] is s[X, Y],
         A is fac/2-1(X),
         B is fac/2-1(Y) !&
         u[s[X, Y], s[A, B]].
f/2-1(X) :-& [X, X].

(defun fac/2-1 (arg#1)
  (if (equal 0 arg#1) 1 (* arg#1 (fac/2-1 (- arg#1 1)))))
fac/2-1(0) :- !& 1.
fac/2-1(X) :- X1 is -(X, 1), Fx1 is fac/2-1(X1), Y is *(X, Fx1) & Y.
rfe-p>
rfe-p> compile
rfe-p>
```

```
rfe-p> fac(5,R)
true
R = 120
rfe-p> f(s[1,2],R)
true
R = u[s[1, 2], s[1, 2]]
rfe-p> f(s[2,3],R)
true
R = [s[2, 3], s[2, 3]]
rfe-p> tripfac(3,R)
true
R = [3, 6]
rfe-p> m
true
R = [4, 24]
rfe-p> m
true
R = [5, 120]
rfe-p> m
unknown
rfe-p> tripfac1(3,F,F1,F2)
true
F = 6
F1 = 24
F2 = 120
rfe-p>
rfe-p> destroy
rfe-p>
rfe-p> az declare( dfmode[ app[g,g,x] ]).
rfe-p> az declare( dfmode[ rev[g,x] ]).
rfe-p>
rfe-p> az app([], X, X).
rfe-p> az app([H|T], X, [H|Y]) :- app(T,X,Y).
rfe-p>
rfe-p> az rev([],[]).
rfe-p> az rev([H|T],X) :- rev(T,T1), app(T1,[H],X).
rfe-p>
rfe-p> undeclare
rfe-p> deta
rfe-p>
rfe-p> listing
rev(X1, X2) :- X2 is rev/2-1(X1).
app(X1, X2, X3) :- X3 is app/3-1(X1, X2).

(defun rev/2-1 (arg#1)
```

```
  (if (equal nil arg#1)
      nil
      (if (consp arg#1)
          (app/3-1 (rev/2-1 (cdr arg#1)) (cons (car arg#1) nil))
          (type-error))))
rev/2-1([]) :-& [].
rev/2-1([H | T]) :- T1 is rev/2-1(T), X is app/3-1(T1, [H]) & X.

(defun app/3-1 (arg#1 arg#2)
  (if (equal nil arg#1)
      arg#2
      (if (consp arg#1)
          (cons (car arg#1) (app/3-1 (cdr arg#1) arg#2))
          (type-error))))
app/3-1([], X) :-& X.
app/3-1([H | T], X) :- Y is app/3-1(T, X) & [H | Y].
rfe-p>
rfe-p> compile
rfe-p>
rfe-p> app([1,2,3],[4,5,6],X)
true
X = [1, 2, 3, 4, 5, 6]
rfe-p> rev([1,2,3,4,5,6],X)
true
X = [6, 5, 4, 3, 2, 1]
rfe-p>
rfe-p> destroy
rfe-p>
rfe-p> az declare( mode[ app[g,g,x] ]).
rfe-p> az declare( mode[ rev[g,x] ]).
rfe-p>
rfe-p> az app([], X, X) :- !.
rfe-p> az app([H|T], X, [H|Y]) :- ! app(T,X,Y).
rfe-p> az app(X,Y,non-list-arg).
rfe-p>
rfe-p> az rev([],[]) :- ! .
rfe-p> az rev([H|T],X) :- ! rev(T,T1), app(T1,[H],X).
rfe-p> az rev(X,non-list-arg).
rfe-p>
rfe-p> undeclare
rfe-p> deta
rfe-p>
rfe-p> listing
rev(X1, X2) :- X2 is rev/2-1(X1).
app(X1, X2, X3) :- X3 is app/3-1(X1, X2).
```

```
(defun rev/2-1 (arg#1)
  (if (equal nil arg#1)
      nil
      (if (consp arg#1)
          (app/3-1 (rev/2-1 (cdr arg#1)) (cons (car arg#1) nil))
          'non-list-arg)))
rev/2-1([]) :- !& [].
rev/2-1([H | T]) :- ! T1 is rev/2-1(T), X is app/3-1(T1, [H]) & X.
rev/2-1(X) :-& non-list-arg.

(defun app/3-1 (arg#1 arg#2)
  (if (equal nil arg#1)
      arg#2
      (if (consp arg#1)
          (cons (car arg#1) (app/3-1 (cdr arg#1) arg#2))
          'non-list-arg)))
app/3-1([], X) :- !& X.
app/3-1([H | T], X) :- ! Y is app/3-1(T, X) & [H | Y].
app/3-1(X, Y) :-& non-list-arg.
rfe-p>
rfe-p> compile
rfe-p>
rfe-p> app([1,2,3],[4,5,6],X)
true
X = [1, 2, 3, 4, 5, 6]
rfe-p> app(3,4,X)
true
X = non-list-arg
rfe-p> rev([1,2,3,4,5,6],X)
true
X = [6, 5, 4, 3, 2, 1]
rfe-p>
rfe-p> destroy
rfe-p>
rfe-p> az declare( mode[ even[g] ] ).
rfe-p>
rfe-p> az even(0) :- !.
rfe-p> az even(X) :- >(X, 1), even(-(X,2)) !.
rfe-p> az even(X) :- <(X,-1), even(+(X,2)).
rfe-p>
rfe-p> az declare( mode[ small[g] ]).
rfe-p>
rfe-p> az small(0).
rfe-p> az small(1).
```

```
rfe-p> az small(2).
rfe-p>
rfe-p> undeclare
rfe-p> deta
rfe-p>
rfe-p> listing

(defun small (arg#1)
  (if (equal 0 arg#1)
      t
      (if (equal 1 arg#1) t (if (equal 2 arg#1) t nil))))
small(0).
small(1).
small(2).

(defun even (arg#1)
  (if (equal 0 arg#1)
      t
      (if (and (> arg#1 1) (even (- arg#1 2)))
          t
          (if (and (< arg#1 -1) (even (+ arg#1 2))) t nil))))
even(0) :- !.
even(X) :- >(X, 1), even(-(X, 2)) !.
even(X) :- <(X, -1), even(+(X, 2)).
rfe-p>
rfe-p> compile
rfe-p>
rfe-p> even(-3)
unknown
rfe-p> even(-2)
true
rfe-p> even(-1)
unknown
rfe-p> even(0)
true
rfe-p> even(1)
unknown
rfe-p> even(2)
true
rfe-p>
rfe-p> small(1)
true
rfe-p> small(3)
unknown
rfe-p>
```

Appendix D

Symbols

- $expr^{\varepsilon}$ is in LL the expression *expr* evaluated:
  `(list 1 2)`$^{\varepsilon}$ = `(1 2)`
- $expr_1 \overset{\varepsilon}{\rightarrow} expr_2$ is equivalent to $expr_1^{\varepsilon} = expr_2$
- $\diamond$ and $\square$ are placeholders on the LLAMA stack and heap used in the description of machine states
- `@`*proc* is the entry address of the procedure *proc*
- $\widetilde{lexpr}$ is the external (printable) LL representation of the LLAMA expression *lexpr* on the stack or heap (where it is in case of a list or structure only a *pointer*)
- $\widehat{lexpr}$ is a copy of *lexpr* in the general purpose memory (Memory) of the LLAMA
- $\overline{lexpr}$ is the LLAMA expression *lexpr* dereferenced
- $\mathcal{G}$ and $\mathcal{X}$ denote the modes *ground* and *unrestricted*, respectively

Bibliography

[Aït-Kaci, 1990] Hassan Aït-Kaci. The WAM: A (Real) Tutorial. Report 5, Digital, Paris Research Laboratory, January 1990.

[Benker *et al.*, 1989] H. Benker, J. Beacco, S. Bescos, M. Dorochevsky, Th. Jeffré, A. Pöhlmann, J. Noyé, B. Poterie, A. Sexton, J.C. Syre, O. Thibault, and G. Watzlawik. KCM: A Knowledge Crunching Machine. May 1989.

[Boley and Richter, 1991] Harold Boley and Michael M. Richter, editors. *Proceedings of the International Workshop on Processing Declarative Knowledge (PDK'91)*, number 567 in Lecture Notes in Artificial Intelligence (LNAI). Springer-Verlag, Berlin, Heidelberg, 1991.

[Boley *et al.*, 1991] Harold Boley, Klaus Elsbernd, Hans-Guenther Hein, and Thomas Krause. RFM Manual: Compiling RELFUN into the Relational/Functional Machine. Document D-91-03, DFKI GmbH, 1991.

[Boley *et al.*, 1993] Harold Boley, Klaus Elsbernd, Michael Herfert, Michael Sintek, and Werner Stein. RELFUN Guide: Programming with Relations and Functions Made Easy. Document D-93-12, DFKI, July 1993.

[Boley, 1986] Harold Boley. RELFUN: A Relational/Functional Integration with Valued Clauses. *SIGPLAN Notices*, 21(12):87–98, December 1986.

[Boley, 1991] Harold Boley. A Sampler of Relational/Functional Definitions. DFKI Technical Memo TM-91-04, DFKI, March 1991. Second, Revised Edition July 1993.

[Boley, 1992] Harold Boley. Extended Logic-plus-Functional Programming. In *Workshop on Extensions of Logic Programming, ELP '91, Stockholm 1991*, LNAI. Springer, 1992.

[Debray and Warren, 1990] Saumya K. Debray and David S. Warren. Towards Banishing the Cut from Prolog. *IEEE Transactions on Software Engineering*, 16(3):335–349, March 1990.

[DeGroot and Lindstrom, 1986] D. DeGroot and G. Lindstrom, editors. *Logic Programming: Functions, Relations, and Equations.* Prentice-Hall, 1986.

[Dorochevsky *et al.*, 1991] Michael Dorochevsky, Jaques Noyé, and Olivier Thibault. Has Dedicated Hardware for Prolog a Future? In H. Boley and M. M. Richter, editors, *Processing Declarative Knowlege*, number 567 in Lecture Notes in Artificial Intelligence, pages 17–31. Springer-Verlag, July 1991.

[Ershov, 1980] A.P. Ershov. Mixed Computation: Potential Applications and Problems for Study. In *Mathematical Logic Methods in AI Problems and Systematic Programming, Part 1*, pages 26–55. Vil'nyus, USSR, 1980. (In Russian).

[Futamura, 1971] Y. Futamura. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

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

[Hall, 1995] Victoria Hall. Integration von Sorten als ausgezeichnete, taxonomische Prädikate in eine relational-funktionale Sprache. Document D-95-04, DFKI Kaiserslautern, March 1995.

[Hanus, 1991] Michael Hanus. Efficient Implementation of Narrowing and Rewriting. In Boley and Richter [1991].

[Haraldsson, 1977] A. Haraldsson. *A Program Manipulation System Based on Partial Evaluation*. PhD thesis, Linköping University, Sweden, 1977. Linköping Studies in Science and Technology Dissertations 14.

[Henderson, 1980] Peter Henderson. *Functional Programming – Application and Implementation*. Series in Computer Science. Prentice-Hall, Englewood Cliffs, NJ, 1980.

[Jones *et al.*, 1989] N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[Kleene, 1952] S.C. Kleene. *Introduction to Metamathematics*. Princeton, NJ: D. van Nostrand, 1952.

[Krause, 1991] 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.

[Kurozumi, 1992] Takashi Kurozumi. Overview of the Ten Years of the FGSC Project. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 9–19, ICOT, Japan, 1992. Association for Computing Machinery.

[Lloyd, 1987] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, Heidelberg, New York, 1987.

[Lock, 1993] Hendrik C. R. Lock. *The Implementation of Functional Logic Programming Languages.* GMD-Bericht Nr. 208. R. Oldenbourg Verlag, 1993.

[McCarthy *et al.*, 1962] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual.* The M.I.T. Press, August 1962.

[Nilsson, 1993] U. Nilsson. Towards a Methodology for the Design of Abstract Machines. *Journal of Logic Programming*, 16(1, 2):163–189, 1993.

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

[Robinson, 1985] J. A. Robinson. Beyond LOGLISP: Combining Functional and Relational Programming in a Reduction Setting, April 1985.

[Sintek, 1991] Michael Sintek. Monolinguistic μCAD2NC: A Deterministic RELFUN Application Generating Abstract NC Programs from CAD-like Geometries. DFKI Kaiserslautern, September 1991.

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

[Sloman and Hardy, 1983] Aaron Sloman and Steve Hardy. POPLOG: A Multi-purpose Multi-language Program Development Environment. AISBQ 47, 1983.

[Steele Jr., 1984] Guy L. Steele Jr. *COMMON LISP: The Language.* Digital Press, March 1984.

[Stein and Sintek, 1991] W. Stein and M. Sintek. RELFUN/X: An Experimental PROLOG Implementation of RELFUN. ARC-TEC Document 91-1, DFKI GmbH, March 1991.

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

[Taylor, 1990] Andrew Taylor. LIPS on a MIPS: Results from a Prolog Compiler for a RISC. In David H. D. Warren and Peter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 174–185, Jerusalem, 1990. The MIT Press.

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

[Warren *et al.*, 1977] David H. D. Warren, Luis M. Pereira, and Fernando Pereira. Prolog - The Language and its Implementation Compared with Lisp. *SIGPLAN Notices*, 12(8):109–115, August 1977. Special Issue.

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

[Wikstrøm, 1987] Ake Wikstrøm. *Functional Programming using Standard ML.* Prentice Hall, 1987.

Index