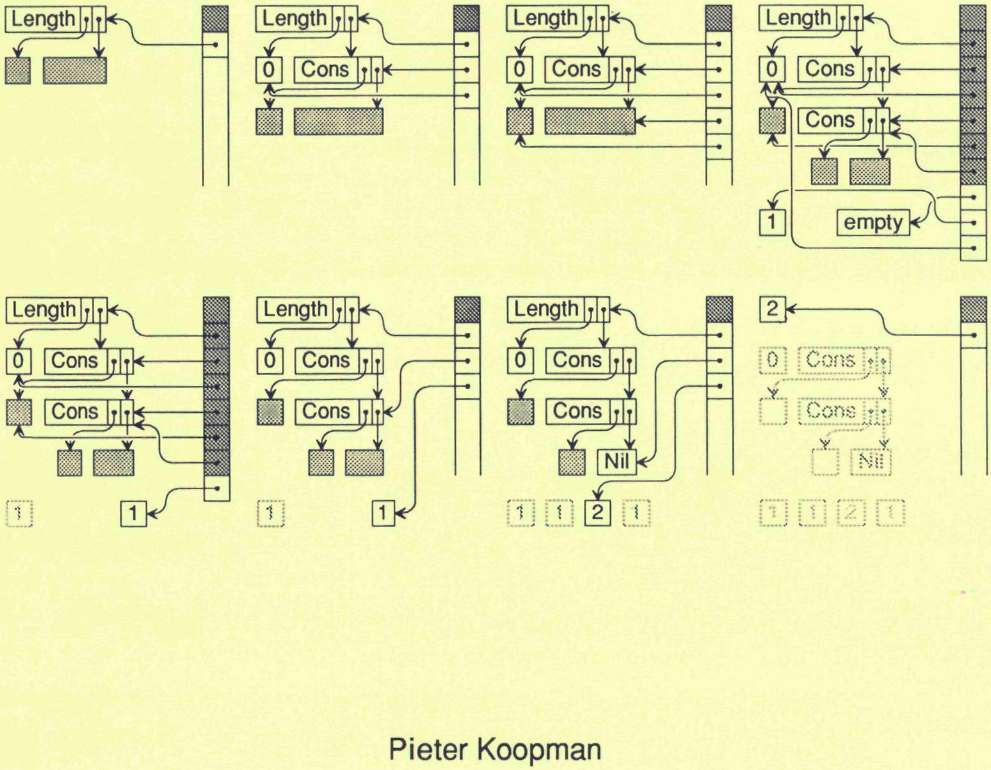
FUNCTIONAL PROGRAMS

AS

EXECUTABLE SPECIFICATIONS



Pieter Koopman

FUNCTIONAL PROGRAMS

AS

EXECUTABLE SPECIFICATIONS

een wetenschappelijke proeve op het gebied van de wiskunde en informatica.

Proefschrift

ter verkrijging van de graad van doctor aan
de Katholieke Universiteit te Nijmegen,
volgens besluit van het college van decanen in het
openbaar te verdedigen op
maandag 10 december 1990
des namiddags te 3.30 uur

door

Petrus Wilhelmus Maria Koopman

geboren op 3 juli 1957 te Nijmegen

Druk: Krips Repro Meppel

Promotor: Prof. dr. H.P. Barendregt
Co-promotor: Dr. ir. M.J. Plasmeijer

ISBN 90-9003689-X

Manuscriptcommissie:

Prof. dr. C.C.A.M. Gielen	Katholieke Universiteit Nijmegen
Dr. ir. S.M.M. Joosten	Techinische Universiteit Twente
Dr. W.G. Vree	Universiteit van Amsterdam

Graag wil ik hierbij iedereen bedanken die op een of andere manier heeft bijgedragen aan de totstandkoming van dit proefschrift. Zonder jullie hulp zou dit proefschrift nooit geschreven zijn.

Table of contents

Chapter 1
Introduction

In 1978 Backus advocated the use of functional programming languages as the new programming discipline needed to solve the problems in software production [Backus 78]. This initiated a large amount of research to achieve at least a reasonable execution speed for these languages. The scope of this research ranges from theoretical studies on the computational models used in the functional languages to the survey and construction of special purpose hardware to execute functional languages in a sequential or parallel way. In the mean time a number of nice languages has been developed. The actual use of functional programming languages received much less attention than the implementation and the study of theoretical properties, partly caused by the bad performance of implementations available.

In this dissertation the suitability of functional programming languages as a vehicle for specifications is investigated. Functional languages seem to be suited as executable specifications due to the high abstraction level and powerful language constructs available. Such a programming language has as advantages over a tailor made specification formalism the well defined semantics and the possibility to check the specifications on consistency by a compiler. Furthermore the specifications are executable. So, the specifications are their own prototype implementation. The suitability of functional languages as specification formalism is illustrated by using them in daily practice on the department of computational models and parallel systems of the Nijmegen university. Sequential and parallel implementations of functional languages are developed here. The underlying computational models of these languages as well as other computational models, like neural networks, are investigated.

In this introductory chapter a motivation of the work presented in this thesis is given. First the function of specifications and prototypes in the construction of software is clarified in section 1.1. Then we argue why functional programming languages are very well suited to be used as a formalism for specifications. Finally, an overview of the specifications presented in this dissertation is given.

1.1 Place of specifications in the implementation process

To clarify the role and importance of specifications in an implementation process, a division in logical phases is made. The implementation process is only one step in a software engineering process. Usually, it is preceded by a study on the kind of product needed and it is accompanied by training and documentation upon its introduction. Afterwards the product still needs maintenance for some time.

The production of an implementation, in hard or software, can be split in a number of phases:

1. *Determination of the requirements*; specification of the problem to be solved.
2. *Design* of an algorithm, that meets these requirements.
3. *Construction* of the product. An implementation of the specified algorithm is made.
4. *Verification* of the product. This phase consists of testing (or proving) whether the product meets the requirements or not. Errors made during the construction are spotted now.

In an ideal situation these phases are strict sequential and the interfaces are unambiguous. But, in many situations these phases are not clearly separable and sequential. The definition of the requirements depends on what is feasible during the implementation and is adjusted as soon as the first tests behave otherwise as expected. Also the interface between the phases is usual not free of misapprehensions. Nevertheless, these phases reflect the four main points in an implementation project; *what* must be solved, *how* must it be solved, *construction* of the product, and its *verification*. In large projects this division in phases can be used for sub-tasks as well, this makes the phase to which an activity belongs depending on one point of view.

Firm specifications are needed to construct an unambiguous interface between the phases. The interface between the construction and verification phase is usually the best defined one. By definition an implementation in a computer language is a formal specification, the program can be checked syntactically and and its behaviour can be studied. In order to prevent misunderstandings the use of formal specifications for the other interfaces is recommended. The specifications occurring in this process are: specification of the requirements, specification of the algorithm and specification of its implementation.

Analysis shows that the verification phase often requires very much time. Not only errors made during the construction of the final product are encountered, but also wrong design decisions or erroneous requirements can be discovered now. In the literature alarming figures are reported; more than half of the errors originates from wrong requirements and over a quarter of the errors in the product are due to the design [Beizer 83, DeMarco 78]. It is usually very expensive to cope with the latter ones. A prototype can be built to prevent this late detection of wrong requirements or an erroneous design.

ad 1: Determination of the requirements

Ideally the functionality of the product is defined in an unambiguous formalism. An ordinary mathematical specification is often used, but such a specification is not always applicable. A number of special purpose formalisms is developed to be used in those situations. In other situations it is not possible to give a precise definition at all. In those situations the requirements must be informally defined, e.g. an abstract machine to express the implementation of graph reduction conveniently.

Sorting items in ascending order is an example where the language of mathematics is very well suited to describe the intended property of the sequence of items, but less suited to describe how such a sorted sequence can be obtained. Provided that proper comparison relations, ≤, are defined on the items of a sequence and their indices one can define:

A sequence S of items $I_1, I_2, \ldots, I_n$ is sorted iff $\forall\ i,j \in \{1,2,\ldots,n\} : I_i \leq I_j \Leftrightarrow i \leq j$.

This specifies fully what is meant by a sorted sequence, but not how it can be constructed from an arbitrary sequence. As far as this definition is concerned a perfect way to obtain the sorted sequence is to generate all possible sequences and pick the first one that obeys the requested property. A constructive specification, an algorithm, is needed to show how a sorted sequence is obtained in a more efficient way. The static specification of the desired properties can be used to prove that the algorithm is correct.

In many situations a formal specification of the required properties cannot be provided, e.g. in general there is not a formal specification of the required properties of a (abstract) machine available. In other situations the proof of the correctness of the algorithm is so elaborated that it is omitted in every-day practice. Given an accurate description of the programming languages A and B and a compilation scheme from A to B, it is in principle possible to prove its correctness, but in practice this proof is usually omitted due to its length and complexity.

ad 2: Design of the algorithm

The design of the algorithm is described by a specification. Experience[1] shows that ambiguities can only be avoided by a formal specification. The design can be the result of an evolution of a sequence algorithms with an increasing algorithmic efficiency.

A specification of the design is a concise description of a constructive algorithm with the following properties:

- *Consistent and correct,* otherwise it is hard to expect a satisfying product.
- *Sufficiently complete,* the specification must contain sufficient information to construct the actual implementation, but matters like execution speed or space requirements are usually not fully determined.
- *Unambiguous,* any doubts about the exact meaning of the specification severely reduce its value, therefore formal languages are required as carrier of the specification.

Using a suited formalism the construction of the formal specification of the algorithm can be helpful in the construction of its design.

When a formal definition of the requirements is available the specification of the algorithm can be proven to be correct, or the correctness of the specification can be assured by deriving the program from the requirements. The length of the proofs and derivations limit the use of these verification techniques to small programs.

A well known constructive algorithm to sort a sequence of items is quicksort [Hoare 62]. This algorithm splits a sequence of items into two parts which are then sorted separately. To split a sequence an element, called median, of the sequence is chosen. The sequence is split into a fraction of elements less or equal to the median and a fraction of elements greater than the median. These fractions are sorted separately. The sorted sequence is obtained by the combination of both sorted fractions. The algorithm terminates by the observation that an empty sequence is always sorted.

$$\begin{array}{lll} \text{quicksort}\ \varnothing & = & \varnothing \\ \text{quicksort}\ S & = & \text{quicksort}\,\{\, e \mid e \in S' \wedge e \le m \,\} \cup \{\, m \,\} \cup \text{quicksort}\,\{\, e \mid e \in S' \wedge e > m \,\}, \\ & & \text{where } S' = S \setminus m \text{ (the sequence } S \text{ without element } m\text{), for some } m \in S \end{array}$$

Here the set notation is used to denote sequences, although the order of elements is relevant here and S might be a multi-set. This specification[2] defines the actions to be performed, but not their order nor the way the elements must be stored.

[1] For instance, it can be shown that the informally specified assignment rule for ALGOL 60 can fail to hold in six ways [Ligler 75].

[2] The clumsy handling of the median is necessary in order to obtain termination for a sequence with some equal elements.

When there is any doubt about the determination of the requirements or the specified algorithm, it makes sense to make one or more prototypes. Prototype implementations can be used to study the dynamic behaviour of the specified system. In this way it can be investigated whether the system will act as required or not.

Prototypes

A prototype is a (partial) implementation to test some key aspects of the intended, probably complex, algorithm. It obeys some aspects of the general specification. The differences between the final product and the prototype implementation are the low efficiency of the prototype, limitations in the amount of data that can be processed and a limited set of the required features. Sometimes a prototype contains additional capabilities to test the specification or to perform some measurements. In order to be useful as prototype it must at least be able to execute small scale tests at a reasonable speed.

The number of prototypes needed depends on the problem. When the requirements and the way to fulfil them are absolutely clear no prototypes are needed. In other cases one or more prototypes are constructed, each one showing some aspects of the product.

Prototypes must be constructed in a high level language for two reasons. First of all the implementation language must be close to the specification language in order to avoid errors in this implementation. When the prototype does not behave as intended it is not clear whether to blame the specification or its implementation. Secondly, the effort needed to construct a prototype must be significantly lower than the amount of work involved in the actual implementation, otherwise it is better to omit the construction of a prototype.

A Miranda[3] prototype of the quicksort algorithm defined above is shown (the sequences are represented by lists and the first element of such a sequence is used as median):

```
quicksort :: [*] -> [*]
quicksort [ ]      = [ ]
quicksort (m : s)  = quicksort [ e | e <- s ; e <= m ] ++ [ m ] ++ quicksort [ e | e <- s ; e > m ]
```

When this algorithm is compared with the specification above only a suited data type to represent sequences and a choice for the median are additionally specified. This data structure happens to be better suited to denote sequences than the set notation. Memory use and order of actions is not specified, only the computations needed are indicated.

[3] Miranda is a trademark of research software Ltd.

ad 3: Construction of the product

After the design of the algorithm and its validation by the prototype, the implementation phase starts. Although a prototype can be a complete implementation it is only in rare cases useful as product. The product must obey several constraints on run time, space consumption and the amount of data to be processed. Also appropriate interfaces to the user and the system must be constructed. Among the topics to be solved are: efficient implementation of the used data structures, choice of the flow of control, efficient implementation of the specified algorithm, modular structure and documentation.

For a product in software also the programming language must be determined. Some of the criteria to chose an implementation language are: the availability of an appropriate implementation; portability; familiarity; suitability for the designed algorithm and speed of execution. When speed is an important aspect of the final product the actual implementation is usually made in an imperative language. Using currently available compiler technology it is hard or impossible to beat imperative languages on most stock hardware. When the speed constraints are less severe and/or more efficient compilers are available other kind of languages can be used for the implementation.

In our sorting example the space requirements and the number of sweeps through the sequence can be improved, also a better choice for the median is feasible. It is possible to specify these matters formally, but usually this is not needed nor wanted. These implementation issues are not essential for the quicksort algorithm, only for a fast special implementation of it.

The product corresponding to quicksort specification depends on the number and the kind of items to be sorted. For a small amount of data the Miranda prototype defined above can directly be used as the final product. But, often speed and space use do matter. Then a real product obeying the specification has to be encoded. An efficient imperative algorithm written in Pascal [Wirth 71] is given below. It sorts the global array a in situ, elements in the array are reshuffled to obtain sets of elements greater and less than the median in an efficient but rather hard to understand way. These sets are recursively sorted.

```
procedure quicksort (l,r: integer);
var m, i ,j, tmp: integer;
begin
   if r > l then                     { sequence not empty                                 }
   begin
      m := a[ r ];                   { median                                             }
      i  := l-1;                     { start for left part; elements less or equal to median      }
      j  := r;                       { start for right part; elements greater or equal to median  }
      repeat
         repeat i := i+1 until a[ i ] >= m;   { expand left part with well placed elements  }
         repeat j := j-1 until a[ j ] <= m;   { expand right part with well placed elements }
         tmp := a[ i ]; a[ i ] := a[ j ]; a[ j ] := tmp;  { swap elements to right part     }
      until j <= i;                           { totally split                               }
      a[ j ] := a[ i ]; a[ i ] := a[ r ]; a[ r ] := tmp;  { median to centre               }
      quicksort(l,i-1);
      quicksort(i+1,r);
   end
end;
```

In this product the memory use and the order of actions are exactly specified. It is optimized for space use and speed; sometimes some superfluous actions are performed in order to avoid a test in all situations.

ad 4: Verification

The correctness of a specification is a serious problem. When some other formal specification is available a correctness proof can be given. For instance a sorting algorithm can be proven to be correct by showing that none of the elements is lost and that the final order of the elements obeys the sorting criterion. When no other specification nor formal requirements of the product are available a correctness proof cannot be given. Usually, correctness proofs appear to be much longer than the specifications, this limits the practical use of proofs to small programs. For large programs proving the correctness is too much work, moreover the value of a proof decreases when its length increases. For very large programs neither testing nor proving can give absolute certainty about the correctness. A proof needs to be correct and complete to prove anything, so the proof itself must be verified. The correctness of a proof is a serious problem, especially with very elaborated proofs. Proofs and testing can only increase the confidence in a product, but both can be very valuable [Joosten 89].

1.2 Specification formalisims

In order to be useful, a specification must be correct and understandable. In general, descriptions are better understandable when they are shorter. Descriptions can be made more compact by omitting irrelevant issues and by the introduction of new description primitives. In order to keep the descriptions clear and easy accessible for many people the introduction of description tools must be done very reluctantly. The brevity of the descriptions is bounded by the expressive power of the description language. In order to be used for an unambiguous definition the specification language itself must be defined properly.

Many special purpose formal description systems are proposed and used nowadays. The advantages of these systems are obvious:

- by the introduction of special purpose notations very compact specifications are possible;
- since these systems are formal they are unambiguous.

However, there are also some disadvantages of the use of special purpose formal systems:

- in order to understand the specifications additional knowledge about the specification language is needed;
- mechanical checks on the syntax and consistency of the specified system are only possible after some implementation of the developed formalism;
- these systems are usually not meant nor suited for execution, so additional work must be done to construct a prototype.

The use of programming languages instead of a special purpose formalism for the specification has a number of attractive advantages.

- The syntactical correctness, absence of unbound identifiers and type consistency can be checked most efficiently and concisely by a compiler. This does not imply that a compiler is able to verify that the complete specification is correct, only partially correctness can be tested.
- Using an existing programming language there is no need nor place for new language constructs in the description. This implicates that the semantics of the description are clear and well defined.
- The specification is its own prototype implementation. This eliminates the effort to construct a prototype and to keep the specified product and prototype similar.
- The specification is checked by the corresponding prototype in a direct way, construction of a separate prototype is only an indirect test of the specification. Although each test shows just that the specified system behaves as shown on the test data, it yields valuable information on the dynamic properties of the system.

The obvious drawback of the use of most programming languages as specification language is the great implementation effort required. Much more time is spent on solving all kind of implementation details than on writing a specification of the product. The difference between the design of the algorithm and the construction of the product vanishes. Due to the implementation details specified in the same description as the algorithm it becomes too elaborate to handle.

Due to the high abstraction level and powerful language constructs functional languages seem to be suited better as specification formalism than imperative programming languages. The comparison of the quicksort algorithm and its prototype implementation in Miranda yields a first indication of the suitability of functional languages as formal specifications. The advantages of using a functional instead of an imperative programming language for the specification are:

- Due to the powerful language constructs and high abstraction level the designer can spend (more of) his time writing the specification instead of solving implementation problems;
- Functional programs are much compacter than equivalent imperative programs, this significantly increases their value as specification.
- Using the abstraction mechanisms of functional languages a hierarchical description can be given. Layered descriptions enable detailed explanations together with a clear overall view.
- When a functional programming language with lazy semantics is used, only the data flow must be specified, the flow of control is deduced automatically from the data needed.

The difference between a functional description and an ordinary functional program is the emphasis on a clear explanation of the system in the description. Efficiency and algorithmic complexity of the description itself are irrelevant. A specification describes an efficient algorithm while an ordinary functional program itself must be efficient. These are just the usual differences between a prototype and the product.

In this thesis the suitability of functional programs as executable specifications is investigated. We are mainly interested in relatively large examples where correctness proofs have a limited applicability due to the size of the problems or to the absence of formal requirements to be met.

A general available functional programming language, Miranda [Turner 85], is used as specification language. Mainly the basic expression rewrite properties of this language are used. This subset is more or less common in all functional languages, so other languages could be used for descriptions as well. Although Miranda is not designed as a specification language, it is reasonably suited. In our opinion it is not worthwhile to design a new tailor made language for each application area, when an existing language is sufficiently fitted. Though Miranda is used with success as a description language, some general remarks concerning its design will be made. This critic concerns the lack of term rewriting semantics for partially parametrized functions. Unfortunately the formal semantics of Miranda has been announced for a long time, but it is still not published. Fortunately, the semantics of a large part of the language is simple. So, a formal specification of the semantics will be close to the language [Fehr 89]. Moreover, only one implementation of Miranda is allowed so it can be used as a, slow, reference implementation.

1.3 Specifications treated in this thesis

A large part of the specifications described in this thesis has actually been used to describe real products in a large implementation project of functional languages on several (parallel) architectures currently under execution at the University of Nijmegen.

As a first step in the implementation process, a functional program is transformed to a graph rewriting system. Graph rewriting systems are a well-defined computational model for functional programming languages [Klop 87, Eekelen 88]. The traditional computational model used for functional programming languages is the λ-calculus [Church 32, Barendregt 84], see for instance [Field 88] for a thorough overview. The advantages of the use of graph rewriting systems as computational model instead of the λ-calculus are: the notions of named (recursive) functions, pattern matching is included in the computational model and graphs enable the detailed description of the sharing of computations.

The second step in the compilation process is the transformation of the graph rewriting system to code for an imperative abstract graph rewriting machine: the ABC-machine. This enables reasoning about implementation issues not expressible in the functional graph rewriting model. The precise flow of control can only be described in an imperative model.

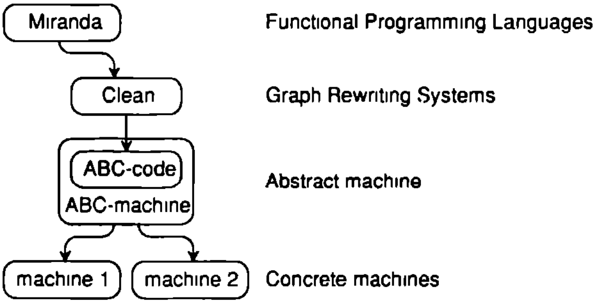


Fig 1.1 The compilation path for functional languages.

The translation of Miranda to Clean and the translation from Clean to ABC-code are specified Miranda. Also the abstract ABC-machine is described in this functional language. In this thesis the specification methods used are described and compared with other descriptions. To show that these description methods can be applied in practice and how comprehensible they are, the specification of the ABC-machine and the translation of Clean to ABC-code are treated.

In chapter 2 it is shown how functional programming languages can be used to describe an (abstract) machine. Such an executable specification is compared with a more common imperative description. The proposed description method is used in the next chapter for a thorough description of the ABC-machine.

A description method for translations is introduced and compared with other descriptions in chapter 4. This description method is employed in chapter 5 to specify the translation of the graph rewriting systems to imperative ABC-code. The translation of Miranda to Clean is described in a similar way

[Koopman 87, Plasmeijer 91], but boils down mainly to the removing of syntactical sugar and is hence less interesting than the translation described here.

Also other aspects of these translations, like type checking [Bakel 90], are described analogously. Currently the implementation of a parallel variant is implemented and described similarly [Smetsers 89, Plasmeijer 91].

Functional specifications can also be used outside the world of functional languages and their implementations. Chapter 6 contains a detailed description of a number of important artificial neural networks. These layered models are constructed of very simple processing elements. Usually these networks are described by a mathematical definition of the input-output relations of the various neurons. The dynamic adaptation of these networks can also adequately be described in a functional language.

Functional languages can be used as programming language or description language in many other fields [Hughes 89], e.g. in [Koopman 88] the use of interactive programs in a functional language is treated. Functional languages can also be used to describe many kinds of systems, for instance digital and analog circuits [Boute 88]. The use of functional programs as specification and prototype derived from a mathematical specification is treated in [Joosten 89].

The last chapter of this thesis contains our conclusions of the suitability of functional programming languages as description languages.

This thesis does not contain an introduction to functional programming. Nowadays there are a number of books containing such an introduction available, we recommend [Bird 88, Plasmeijer 91, or Field 88] for a thorough introduction. The first book is one of the few books treating functional languages without treating their implementation. The second book uses Miranda as functional language and shows that graph rewriting systems are a very suited computational model for the (parallel) implementation of functional languages. The last book uses Hope [Burstall 80] as functional language and uses the λ-calculus as computational model for the implementation. The language Clean is described only informally in chapter 5 of this thesis. A number of descriptions is available in the literature [Brus 87, Barendregt 87b, Eekelen 89, Plasmeijer 91].

Chapter 2
Operational Machine Description

In this chapter functional programming languages are used to describe (abstract) machine architectures. For each machine there are many possible descriptions, each one showing different aspects of the machine, e.g. the instruction repertoire, the micro-programming level or the hardware. The intention of the description presented in this chapter is to show the components and to explain their interactions as relevant from a programmers point of view.

The machine description consists of an implementation of the machine in a functional language. This implementation is not meant to be efficiently executable, but to supply a clear view of the relevant machine aspects. The description of machines in a functional programming language has a number of advantages over conventional descriptions. The description is an ordinary functional program. This implies that the description is executable and serves as a prototype machine implementation. The provided implementation appears to be valuable to study the dynamic behaviour of the machine. Using this prototype implementation the machine can be run in an early state.

Using the abstraction mechanisms of the functional programming language a hierarchical machine description can be given. Such a layered machine description enables a detailed explanation, without getting an overall view glutted by too much details at the top level of the description. The description can be chosen such that the machine aspects of prime interest are clearly shown, while other aspects are hidden. In this way one can abstract from the implementation of some machine components on one level, but accurately specify that object on another level. This hierarchical description method introduced in this chapter is generally applicable. It can also be used to describe large concrete machines, although they tend to have more parts than the machines shown in this thesis, hence they have a more elaborate state.

Due to the use of a functional programming language as description formalism, the descriptions are clearer and less error prone than conventional descriptions. The descriptions give valuable guide-lines to achieve an efficient implementation and serve as reference implementation of the described machine.

Commonly used description methods for (abstract) machines are:

- textual description, these specifications are hard to get unambiguous and correct;
- by the state changes in an imperative fashion, see this chapter for a comparison;
- by a definition of the semantics of the instruction set, see chapter 4 and 5 for a discussion of semantical descriptions.

After the introduction of the description method in section 2.1, a small but complete example is given to illustrate the description method proposed here. The example is taken from a textbook of computer architecture where it is used to illustrate the concept of micro-programming. Here only the instructions of the conventional machine level are specified to illustrate the proposed description method. The micro-programming level can be described similarly. In the next chapter this description method is used to specify an abstract graph rewrite machine as a large and more complex example.

2.1 The description method

The machine descriptions introduced in this chapter primarily define the semantics of the machine instructions. In general such a specification is not in close correspondence with possible hardware implementations of the machine described. This implies that the description usually does not reflect the common von Neumann computer architecture (a Central Processing Unit (CPU) and a memory interconnected by a bus), but shows the the machine components relevant from the programmers point of view. All machine components together determine the state of the machine. Instructions change the machine state. The machine descriptions introduced here, will be based on a specification of the state of the machine.

The instructions are described in a two level model. On the top level, the **instruction level**, the exact state transition of each instruction is specified in terms of micro-instructions. The **micro-instructions** form the bottom level of the description and define primitive operations on the logical components of the machine.

The machine state

The machine description method presented in this chapter is applicable to sequential, imperative machines. Such a machine consists of a number of constructing parts (like various pieces of memory and input/output channels). The state of the machine is completely described by the contents of the constructing parts. The state and hence the contents of these parts can be manipulated by the instructions defined on the machine.

For the description, the complete state of the machine is recorded in a data structure, called **state**. For every component of the machine a suited (abstract) data type is used, to describe its status or contents.

For instance, the state of a machine with one register (reg), a program counter (pc), a stack (stack) and a memory (mem) is described by:

```
(pc, reg, stack, mem)
```

The definition of the state affects the architecture as it is presented to the user. For example, when it is important to show that the stack is maintained with a stack pointer (sp) in the global memory (mem), the used machine architecture must be slightly different. This is reflected in the state of this machine, it is described by:

```
(pc, reg, sp, mem)
```

These two machine models can be depicted as:

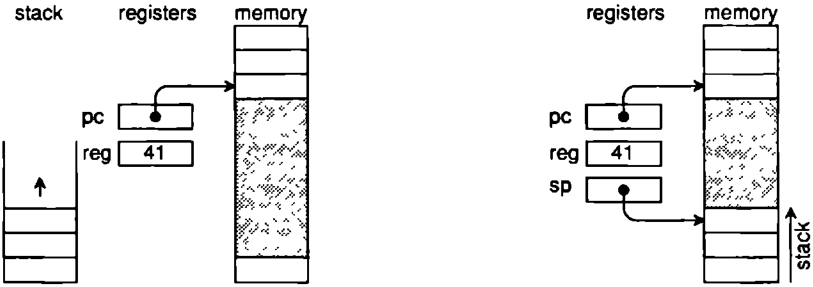


Fig 2.1 The machine with a stack, a program counter, one register and a memory is depicted on the left. On the right hand side a similar machine is drawn, but here the stack is maintained with a special register containing a stack pointer in the memory.

It is not possible to use the state corresponding to the left machine to describe the right architecture. Adjustable data structures, like memories and stacks, must always be constructed in a tree-like structure. This shows clearly the dependencies between the machine components, which is generally desirable.

The micro-instructions

To access the data structures describing the components of the machine tailor made functions are used. These functions are called *micro-instructions*, since they will form the building blocks of machine instructions. These micro-instructions define an abstract data type describing the machine component at hand, e.g. a stack can be described by the functions to push and pop elements and to access an element on the stack. Only in extremely simple cases the use of the abstract data types can be omitted, e.g. registers with ordinary integer arithmetic can sometimes be modelled by the language implementation of integers.

A set of micro-instructions defining the stack for the first example machine architecture above is:

```
abstype  stack
  with   s_top  :: stack -> value
         s_pop  :: stack -> stack
         s_push :: value -> stack -> stack
```

Micro-instruction names are usually prefixed with some characters indicating the data structure manipulated.

A set of micro-instructions defining the memory is:

```
abstype  mem
  with   get_num  :: address -> mem -> num
         get_ins  :: address -> mem -> instruction
         m_update :: address -> num -> mem -> mem
```

To create an executable specification all machine components have to be implemented. The implementation of the data type can be kept simple when the structure of the associated machine part is not a topic of interest in the given description, e.g. a stack can be represented as a list. The data type can also show a more accurate model of the machine component, e.g. the stack can also be represented by an array of values and an index in that array to represent the stack pointer.

Other micro instructions used in this introduction have similar definitions. A complete set of micro-instructions is defined in section 2.2 and in the next chapter.

The instructions

Instructions and their execution can be modelled in different ways. The first possibility resembles denotational semantics; a function which interprets the current instruction is constructed. The actions of this function are in close correspondence with the instruction cycle of the CPU. The current instruction is fetched from the memory and the machine state is adjusted accordingly. In order to use pattern matching on the kind of the current instruction at the top level, it must be at a fixed place, therefore the program is usually modelled as a list of instructions. The head of this list is the current instruction. The possibility we propose here uses a more realistic memory model and the instructions change the state of the machine themselves.

In this description the instructions are functions that take the current machine state as argument and deliver the new state, as changed by the instruction. So an instruction is in the description a function of type state -> state. Hence, the general format of an instruction is *mnemonic instruction-arguments state = new-state*. The state transformation is described in terms of the micro-instructions introduced above. Although it is possible to write complex expressions in the instruction specifications, this is not recommended. To obtain a clear specification the new machine state yielded by the instruction is specified using only micro instructions, arguments and constants. To enhance the readability locally defined identifiers are used in the construction of the machine state.

A jump to subroutine instruction with the corresponding return instruction for the machine with a stack as introduced above can be described as:

```
jsr :: address -> instruction
jsr address (pc, reg, st, mem)
   = (pc', reg, st', mem)
     where  pc'   = address
            st'   = s_push pc st

rtn :: instruction
rtn (pc, reg, st, mem)
   = (pc', reg, st', mem)
     where  pc'   = s_top st
            st'   = s_pop 1 st
```

We prefer to specify the instructions as above instead of the shorter equivalent given below, since it is better readable.

```
jsr :: address -> instruction
jsr address (pc,reg,st,mem) = (address,reg,s_push pc st,mem)
```

A jump to subroutine and corresponding return instruction for the machine with a stack maintained with a stack pointer in the memory are:

```
jsr :: address -> instruction
jsr address (pc, reg, sp, mem)
   = (pc', reg, sp', mem')
     where  pc'   = address
            sp'   = sp_dec sp
            mem'= m_update sp' pc mem
```

```
rtn :: instruction
rtn (pc, reg, st, mem)
   = (pc', reg, sp', mem)
     where  pc'  = get_num sp mem
            sp'  = sp_inc sp
```

Program execution

The instruction sequence to be executed, the program, is also part of the machine. The program is stored in the memory and the instruction to be executed, the current instruction, is addressed by a **program counter**. It can be executed by extracting the instruction from the memory and applying it to the current state. This clearly reflects the von Neumann machine instruction cycle.

There are two possible places to increment the program counter in order to make the next instruction the current instruction after one instruction cycle. It is possible to increment the program counter in each instruction explicitly, or it can be done in the instruction cycle. Since most machine models use the latter approach we have adopted it here. An instruction cycle executing a single instruction of the machine described above is modelled by:

```
instruction_cycle :: state -> state
instruction_cycle (pc, reg, st, mem)
   = current_instruction (pc', reg, st, mem)
     where  pc'                 = pc_next pc
            current_instruction = get_ins pc mem
```

Note that the function instruction_cycle is not a machine instruction, although it has the same type as an instruction. It is just a function that executes the current instruction that happens to have the same type as instructions.

The instruction cycle above executes just one instruction, in most situations it is more convenient that a machine continuously executes instructions. This can be achieved by applying the instruction cycle recursively to the state delivered by the current instruction.

```
instruction_cycle :: state -> state
instruction_cycle (pc, reg, st, mem)
   = instruction_cycle (current_instruction (pc', reg, st, mem))
     where  pc'                 = pc_next pc
            current_instruction = get_ins pc mem
```

A complete machine description consists of the definition of its state, and the definition of all instructions able to alter this state. The use and purpose of defining the machine are not given by the machine description. Many abstract machines are defined with a very specific purpose, this aim cannot be deduced from a bare machine description, so usually there must be also a description of the objective of the abstract machine to give it any sense.

2.2 Description of a simple machine

To compare the functional description with a conventional description we present two descriptions of a simple machine. First, the conventional description will be shown. Then the functional specification is given. In chapter 3 an elaborated abstract machine description is treated.

The machine presented here is a simple conventional machine, called Mac-1. Mac stands for macroarchitecture. It is used by Tanenbaum [Tanenbaum 84] to illustrate the concept of micro-programming.

The Mac-1 is a small machine with a memory of 4096 16-bit words. This memory contains the program to be executed and a stack. The stack grows from high memory addresses to lower ones. The top of the stack is indicated by the stack pointer (sp). The current instruction is indicated by the program counter (pc). The machine has one register, the accumulator (ac), to store the result of computations etc. The architecture of Mac-1 can be depicted as:

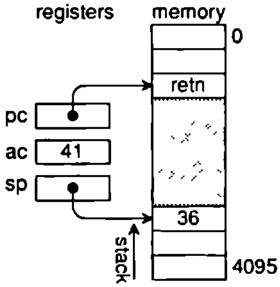


Fig 2.2 The architecture of Mac-1

Four addressing modes are provided in this machine:

immediate: the operand is specified in the low order bits (8 or 12) of the instruction;

direct: the low-order 12-bits of the instruction are the address of the operand;

indirect: the accumulator contains the address of the operand;

local: the operand is on the stack, the offset is given in the low-order 12-bits of the instruction.

The addressing mode is indicated by the instruction used. There is a very limited set of instructions available:

Load: load the accumulator with the specified operand;

Store: store the contents of the accumulator at the specified address in the memory;

Add: add the operand to the contents of the accumulator, the sum is stored in the accumulator;

Sub: subtract the operand from the contents of the accumulator; the result is left in the accumulator;

Jump: set the program counter to the specified address conditional to the contents of the accumulator;

Push: push the specified operand on the stack;

Pop: pop an item from the stack;

Swap: exchange the contents of the accumulator and stack pointer;

Sp handling: *increment* or *decrement* the stack pointer by the 8-bit constant given in the instruction.

We presume that the reader is sufficiently familiar with conventional machine architectures to understand the description and use of the instructions. The instruction set is specified in the figure below. The meaning of the instructions is given by a Pascal fragment. In these fragments, m[x] refers to memory word x.

Binary	Mnemonic	Instruction	Meaning
0000xxxxxxxxxxxx	LODD	Load direct	ac= m [x]
0001xxxxxxxxxxxx	STOD	Store direct	m [x] = ac
0010xxxxxxxxxxxx	ADDD	Add direct	ac = ac + m [x]
0011xxxxxxxxxxxx	SUBD	Subtract direct	ac = ac - m [x]
0100xxxxxxxxxxxx	JPOS	Jump positive	**If** ac ≥ 0 **then** pc = x
0101xxxxxxxxxxxx	JZER	Jump zero	**If** ac = 0 **then** pc = x
0110xxxxxxxxxxxx	JUMP	Jump	pc = x
0111xxxxxxxxxxxx	LOCO	Load constant	ac = x (0 ≤ x ≤ 4095)
1000xxxxxxxxxxxx	LODL	Load local	ac = m [sp + x]
1001xxxxxxxxxxxx	STOL	Store local	m [x + sp] = ac
1010xxxxxxxxxxxx	ADDL	Add local	ac = ac + m [sp + x]
1011xxxxxxxxxxxx	SUBL	Subtract local	ac = ac - m [sp + x]
1100xxxxxxxxxxxx	JNEG	Jump negative	**If** ac < 0 **then** pc = x
1101xxxxxxxxxxxx	JNZE	Jump non zero	**If** ac ≠ 0 **then** pc = x
1110xxxxxxxxxxxx	CALL	Call procedure	sp = sp - 1 , m [sp] = pc , pc = x
1111000000000000	PSHI	Push indirect	sp = sp - 1 , m [sp] = m [ac]
1111001000000000	POPI	Pop indirect	m [ac] = m [sp] , sp = sp + 1
1111010000000000	PUSH	Push onto stack	sp = sp - 1, m [sp] = ac
1111011000000000	POP	Pop from stack	ac = m [sp], sp = sp + 1
1111100000000000	RETN	Return	pc = m [sp], sp = sp + 1
1111101000000000	SWAP	Swap ac,sp	tmp = ac , ac = sp , sp = tmp
11111100yyyyyyyy	INSP	Increment sp	sp = sp + y (0 ≤ y ≤ 255)
11111110yyyyyyyy	DESP	Decrement sp	sp = sp - y (0 ≤ y ≤ 255)

xxxxxxxxxxxx is a 12-bit machine address; in column 4 it is called x.
yyyyyyyy is an 8-bit constant; in column 4 it is called y.

The Mac-1 instruction set as described by Tanenbaum.

The functional specification presented below is merely a substitute for the second and last column. The mapping of binary numbers to instructions is not difficult, but omitted for reasons of brevity.

The state of Mac-1

As shown above the architecture of Mac-1 contains three registers (pc, ac and sp) and a memory. The state of Mac-1 is fully determined by the contents of the three registers and the memory. In our description the state is given by the tuple (pc,ac,sp,mem).

```
state       ==  (pc,ac,sp,mem)
instruction ==  state -> state
```

The micro-instruction level of Mac-1

The registers of Mac-1 contain 16-bit integers. Just like the original description we will not use 16-bit arithmetic for the description. This is perfect to obtain a small and simple description, but the numbers are a super set of the 16-bit integers. These registers are modelled by Miranda numbers. When it is required that the content of the registers is limited to 16-bit integers a tailor made arithmetic must be used. In our description all arithmetical operators must be replaced by a proper micro-instruction. Here, no tailor made micro-instructions are needed for the registers, they can be manipulated just like all other numbers in Miranda.

```
pc   ==  num
ac   ==  num
sp   ==  num
```

The memory is defined by a small set of micro-instructions. The four access functions defined are:

get a: returns the number stored at address a in the memory;

get_ins a: returns the instructions stored at the indicated address in the memory;

update a n: replaces the contents of memory location with address a with the number n;

store prog: stores the complete program prog in the memory.

The memory is formally defined by the abstract type mem.

```
abstype  mem
  with   get     :: address -> mem -> num
         get_ins :: address -> mem -> instruction
         update  :: address -> num -> mem -> mem
         store   :: program -> mem
```

These micro-instructions are of the same complexity as the Pascal notations, such as m[x], used in the previous description. The informal description and type definitions are sufficient to understand them. Though the definition will be seldomly referenced it is included for the sake of completeness and to show that such an implementation is indeed very simple.

The implementation of the abstract type mem can be very simple. We have chosen the description such that it is an error to use instructions as data and vice versa. This enables the storing of instructions directly in the memory, rather than to use numbers which must be decoded to instructions. the micro-instruction definitions shown here have additional rule alternatives in order to give better error messages in erroneous situations. In this way the value of the prototype implementation is increased, these error reporting alternatives are not regarded as a part of the specification which must be obeyed by every implementation.

```
program    ==   [word]
address    ==   num
mem        ==   [word]

word       ::=  I instruction   |
                N num           |
                Undef

get      address mem =  w_num (get_w address mem)
get_ins  address mem =  w_ins (get_w address mem)

get_w gaddress -> mem -> word
get_w 0 (w:ws)   =  w
get_w n (w:ws)   =  get_w (n-1) ws
get_w n [ ]      =  error "illegal memory reference"

w_ins :: word -> instruction
w_ins (I i)      =  i
w_ins (N n)      =  error "treating a number as instruction"
w_ins Undef      =  error "taking a instruction from an undefined word"

w_num :: word -> num
w_num (N n)      =  n
w_num (I i)      =  error "treating an instruction as data"
w_num Undef      =  error "taking a number from an undefined word"

store instructions = instructions ++ rep (4096 - # instructions) Undef

update address new mem = update_w address (N new) mem

update_w :: addres -> word -> mem -> mem
update_w 0 new (w:ws)  = new:ws
update_w n new (w:ws)   = w : update_w (n-1) new ws
update_w n new [ ]      = error "illegal memory update"
```

The Mac-1 instructions

A direct transformation of the meaning of the instructions given above into the proposed description frame work is:

```
lodd x (pc,ac,sp,me)
 = (pc,ac',sp,me)
   where  ac'   =  get x me
stod x (pc,ac,sp,me)
 = (pc,ac,sp,me')
   where  me'   =  update x ac me
addd x (pc,ac,sp,me)
 = (pc,ac',sp,me)
   where  ac'   =  ac + get x me
subd x (pc,ac,sp,me)
 = (pc,ac',sp,me)
   where  ac'   =  ac - get x me
jpos x (pc,ac,sp,me)
 = (pc',ac,sp,me)
   where  pc'   =  cond (ac >= 0) x pc
jzer x (pc,ac,sp,me)
 = (pc',ac,sp,me)
   where  pc'   =  cond (ac = 0) x pc
jump x (pc,ac,sp,me)
 = (pc',ac,sp,me)
   where  pc'   =  x
loco x (pc,ac,sp,me)
 = (pc,ac',sp,me)
   where  ac'   =  x
lodl x (pc,ac,sp,me)
 = (pc,ac',sp,me)
   where  ac'   =  get (sp + x) me
stol x (pc,ac,sp,me)
 = (pc,ac,sp,me')
   where  me'   = update (sp + x) ac me
addl x (pc,ac,sp,me)
 = (pc,ac',sp,me)
   where  ac'   =  ac + get (sp + x) me
subl x (pc,ac,sp,me)
 = (pc,ac',sp,me)
   where  ac'   =  ac - get (sp + x) me
jneg x (pc,ac,sp,me)
 = (pc',ac,sp,me)
   where  pc'   =  cond (ac < 0) x pc
```

```
jnze x (pc,ac,sp,me)
 = (pc',ac,sp,me)
   where  pc'   =  cond (ac ~= 0) x pc
call x (pc,ac,sp,me)
 = (pc',ac,sp',me')
   where  pc'   =  x
          sp'   =  sp - 1
          me'   =  update sp' pc me
pshi (pc,ac,sp,me)
 = (pc,ac,sp',me')
   where  sp'   =  sp - 1
          me'   =  update sp' ac me
popi (pc,ac,sp,me)
 = (pc,ac,sp',me')
   where  sp'   =  sp + 1
          me'   =  update ac top me
          top   =  get sp me
push (pc,ac,sp,me)
 = (pc,ac,sp',me')
   where  sp'   =  sp - 1
          me'   =  update sp' ac me
pop (pc,ac,sp,me)
 = (pc,ac',sp',me)
   where  sp'   =  sp + 1
          ac'   =  get sp me
retn (pc,ac,sp,me)
 = (pc',ac,sp',me)
   where  pc'   =  get sp me
          sp'   =  sp + 1
swap (pc,ac,sp,me)
 = (pc,ac',sp',me)
   where  ac'   =  sp
          sp'   =  ac
insp y (pc,ac,sp,me)
 = (pc,ac,sp',me)
   where  sp'   =  sp + y
desp y (pc,ac,sp,me)
 = (pc,ac,sp',me)
   where  sp'   =  sp - y
```

Execution of Mac-1 programs

In order to execute a Mac-1 program it must be stored in the memory. The program execution is chosen to start at address 0. A real Mac-1 machine must run forever, but a simulation must terminate. The simulation is aborted when the program counter becomes negative. We have chosen to show the program counter, the stack pointer and the contents of the accumulator in the final machine state.

```
run :: program -> (pc,ac,sp)
run program
   = instruction_cycle (pc,ac,sp,mem)
     where pc   = 0
           ac   = 0
           sp   = 0
           mem  = store program

instruction_cycle :: state -> (pc,ac,sp)
instruction_cycle (pc,ac,sp,mem)
   = instruction_cycle (instr (pc',ac,sp,mem))    , if pc >= 0
   = (pc,ac,sp)                                    , otherwise
     where pc'   = pc + 1
           instr = get_ins pc mem
```

It is easy to write an instruction cycle that delivers trace information about the executed program. The trace information shown after each instruction execution could consist of the current contents of the registers and the top of the stack.

Unfortunately it is not possible to show the current instruction in the trace. The instructions stored in the memory of Mac-1 are partially parametrized functions. The Miranda system[1] shows all partially parametrized functions as `<function>` and the comparison of such curried functions causes an erroneous program termination with the message `program error: attempt to compare functions`. This behaviour is suggested by the computational model used; the λ-calculus. In the λ-calculus the usual semantics of a comparison of two functions is a test for extensional equality: do these functions have the same reduct for all possible arguments. This is in general an undecidable property.

For curried functions like the instructions of Mac-1 the reduction behaviour of these partially parametrized functions is irrelevant. We are interested in the syntactical value of these expressions. In such a setting two functions are equal if their normal forms are syntactically equal. A test for syntactic equality is part of the semantics of rewrite systems. Such a syntactical comparison can be added to the λ-calculus, it is known as Church's δ rule; λ-calculus with this

[1] The Miranda system version 2.009 (13 November 1989) of Research Software Ltd. is referenced in this thesis as 'the Miranda interpreter'. All timings are done on a SUN-3/280.

extension is sound, but Church's δ rule itself is not λ-definable [Barendregt 84]. Church's δ rule will not solve our problem. We want to show the function name and not in the corresponding λ-term which is the normal form of that function name. Moreover, the λ-term of an recursive function has not a normal form.

The rewrite semantics is obviously the one needed here for the partially parametrized functions. It would be better if the semantics of functional programming languages with pattern matching was based on rewrite systems. (Concurrent) Clean is a functional programming language based on term graph rewriting that allows the suggested use of partially parametrized functions [Brus 87, Eekelen 89].

An example Mac-1 program

To illustrate the possibility of program execution on the given Mac-1 specification an implementation of the nfib function in Mac-1 code is shown below. The result of this function is equal to the number of function calls needed to compute it. The nfib-number is obtained by dividing the result by the number of seconds needed to compute it, i.e. the number of function calls per second. The Miranda definition of the nfib function reads:

```
nfib 0   =  1
nfib 1   =  1
nfib n   =  1 + nfib (n-1) + nfib (n-2)
```

The implemented function nfib expects its argument in the accumulator and also leaves its result in that register. The function implementation is straightforward. The evaluation of the function call nfib 5 on Mac-1 can be simulated by reducing the expression run (nfib 5) in Miranda.

```
nfib :: num -> program
nfib n
= [  I  ( loco   100  ) ,  || 0:         initial stack pointer
     I  ( swap        ) ,  || 1:         load stack pointer
     I  ( loco   n    ) ,  || 2:         main ; load argument for nfib
     I  ( call   5    ) ,  || 3:         call nfib
     I  ( jump   stop ) ,  || 4:         stop machine

     I  ( jnze   8    ) ,  || 5: nfib:   jump to nfib2 if n ≠ 0
     I  ( loco   1    ) ,  || 6:         n = 0; load accumulator with result
     I  ( retn        ) ,  || 7:         done; return to caller.

     I  ( subd   22   ) ,  || 8: nfib2:  compute n-1
     I  ( jnze   12   ) ,  || 9          jump to nfib2 if n ≠ 1
     I  ( loco   1    ) ,  || 10:        argument = 1; load accumulator with result
     I  ( retn        ) ,  || 11:        done, return to caller
I  ( push       ) ,  || 12: nfib3 save n-1 on the stack
I  ( subd  22   ) ,  || 13:       compute n-2
I  ( call  5    ) ,  || 14:       compute nfib (n-2)
I  ( push       ) ,  || 15:       save nfib (n-2) on the stack
I  ( lodl  1    ) ,  || 16:       load n-1 into accumulator
I  ( call  5    ) ,  || 17:       compute nfib (n-1)
I  ( addl  0    ) ,  || 18:       add the computed nfib values
I  ( addd  22   ) ,  || 19:       add 1
I  ( insp  2    ) ,  || 20:       clean up stack
I  ( retn       ) ,  || 21:       done; return to caller

N  1              ]  || 22:       The constant 1.
```

The shown program notation with addresses is rather cumbersome to use. It is more convenient to use an assembler level where labels can be used instead of numbers, also the constructors to distinguish numbers and instructions can be generated by the assembler. In the next chapter such an assembler level is introduced for the ABC-machine described there.

The Miranda interpreter executes circa 20 Mac-1 instructions each second. The resulting nfib-number is 2. This is not extremely fast, but sufficient to execute small Mac-1 programs. This execution of Mac-1 programs is certainly much faster and more accurate than the pencil and paper simulations necessary with the first specification.

2.3 Discussion

The description of the instruction set of Mac-1 in Miranda is a bit longer than the original description given in the table. This is caused by the layout rules we have imposed ourselves to obtain the clearest description. Instructions can also be described in a single line as shown in Section 2.1. Also the binary representation of instructions is not shown in the functional description. Nevertheless, very clear and compact specifications are obtained using this description method. Due to the two level description the operational semantics of each instruction are clearly specified in terms of simple micro-instructions.

The micro-instructions are simple access functions defining the machine components. By adjusting the micro-instruction level any amount of detail can be incorporated in these machine components. The machine description shown here can be extended to perform 16-bit arithmetic and allow the use of instructions in the memory as data and vice versa by changing only the micro-instruction level.

The use of a functional programming language as description formalism has a number of advantages:

- The description can be verified by the compiler for the functional language and has well defined semantics.

- The specification is more complete. For instance, in the functional description it is clear when the program counter is incremented, and that the incremented program counter is stored on the stack in a call instruction.
- The description is more direct. There are no temporary variables needed, as used in the imperative description of the swap instruction. This is an advantage of having an explicit state in the description of the instructions instead of an implicit global state.
- The description serves as a prototype implementation of the machine and is able to execute programs. When programs execute correctly on the prototype, this increases the confidence in the correctness of the description and the suitability of the specified machine.

It is a pity that Miranda lacks rewrite semantics, this limits the use of the specification to trace the execution of programs.

In order to use the specification for the execution of large programs an assembler level is needed.

The functional descriptions require, just like many other description methods, a single locus of control and a tree-like structure for the state. When a machine does not possess this structure the actual structure must be described. For instance, a parallel machine description contains a scheduler to model the concurrent execution. In this way a parallel architecture is modelled by a sequential machine and the description method can be applied, although there is a lot more description overhead.

Chapter 3
The ABC-Machine

This chapter presents a successful application of the operational machine description method introduced in the previous chapter for a large and complex concrete problem. The development of this specifcation was a test of the suitabilty of the description method. In order to be useful this description method must yield specifications which are: a clear definition for the product to construct; a useful prototype implementation; fast and easy to develop.

The ABC-machine is an imperative abstract machine architecture for graph rewriting. To enable a detailed description of the graph rewriting process a huge number of instructions is defined. Circa 50 instructions are used to describe the rewriting process, even a larger number of instructions is provided to handle basic values. The letters A, B and C represent the three stacks used in this machine. This virtual machine is designed to describe graph reduction on a low level of abstraction, close to the level of a concrete machine architecture. This machine is an abstraction of the large class of concrete stack based machine architectures. ABC-machine code is used as intermediate level on the compilation path of functional languages to concrete machine code.

The first step on this translation path consists of the translation of functional languages to the graph rewriting language Clean [Eekelen 89]. In this translation step the syntactical sugar of functional languages is removed and sharing of computations is specified accurately. In many cases the translations from a functional program to an equivalent Clean program are straightforward.

In some cases, like guards and ZF-expressions, the transformations become tedius and rather complex. These transformations have also been formally specified in a functional language [Koopman 88]. In the next step Clean is translated to ABC-code. This translation links the world of graphs and strategies to a sequence of imperative instructions and is described in chapter 5. Finally, ABC-instructions are translated to, or interpreted by, concrete machine code. During this last step all details of the concrete machine become important to achieve an efficient implementation. A simple implementation of the ABC-machine is relatively easy to make, for instance using macro expansion.

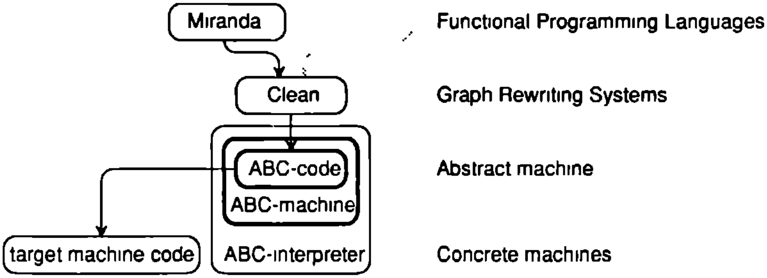


Fig 3.1 The place of the ABC-machine in the translation process of functional languages.

There are several reasons to introduce the additional intermediate level defined by the abstract machine on the compilation path. First of all, the abstract machine helps to get a more structured implementation. When the abstraction is well designed, irrelevant machine dependent issues are omitted. The trade-offs in the imperative graph rewriting process are shown clearly. Some of the restrictions imposed by a concrete machine, such as limited resources (like the limited stack size) are not necessarily present in an abstract machine. Of course, all problems have to be solved on one level or the other. However, the separation of concerns helps to get a more structured view on the problems (and their solutions) one has to face when making an implementation of functional languages.

Another advantage of the additional intermediate level is that the portability of the implementation is increased. To implement functional languages on a new concrete machine only a new ABC-machine has to be implemented. Since the ABC-machine is an imperative machine on a relatively low level of abstraction, it is much easier to implement directly than Clean or a high level functional language.

It is not feasible to use an existing imperative programming language (like C or Pascal) instead of ABC-code since it imposes too much restrictions on the language constructs possible, hence the execution speed will be too low. So, a tailor made abstract machine is developed.

In the next section a global overview of the architecture of the ABC-machine is given by specifying its state. In section 3.2, the basic components of the machine are introduced by the definition of the micro-instruction set. The actual ABC-machine instructions are specified in terms of these micro-instructions in section 3.3. How the abstract machine executes an ABC-program is explained in section 3.4. In section 3.5 an assembler form of the ABC-instructions is introduced to improve the readability and to simplify the generation of ABC-programs. In section 3.6 the ABC-machine is compared with the well-known G-machine [Johnson 84], the machines themselves and the description methods are both contrasted. Finally, the ABC-machine and its description method are briefly discussed.

3.1 The architecture of the ABC-machine

The main part of the instruction set of the ABC-machine is designed to conveniently express graph rewriting. When this instruction set is mapped on a traditional concrete machine the resulting code will be executed relatively slow, since present-day architectures are not at all designed for graph rewriting. A graph structure is not directly available on these architectures, but it has to be represented by some data structure. The modification of the graph will lead to complex memory management problems, involving garbage collection. For many simple calculations the use of graphs is not effective. This observation has triggered the introduction of the second part of the abstract machine. This part is an abstraction from a traditional stack-based architecture. To obtain an efficient program these fast instructions are used and graph rewriting is avoided whenever possible!

The ABC-machine consists of the following memories:

- the *A(rgument)-stack* used to reference nodes in the graph store;
- the *B(asic value)-stack* used to hold and manipulate basic values efficiently;
- the *C(ontrol)-stack* to store and retrieve return addresses;
- the *graph store* containing the graph to be rewritten;
- the *descriptor store* containing information about the symbols used;
- the *program counter* contains an identification of the instruction (instrid) to be executed;
- the *program store* containing the instruction sequence to be executed;
- an *i(nput)-o(utput) channel* to enable interaction with the world.

This machine can be depicted[1] as:

[1] Note: not all pointers are drawn; The symbol names and labels in the nodes are actually pointers in the descriptor store and the program store. Using arrows for these pointers would make the picture unreadable.

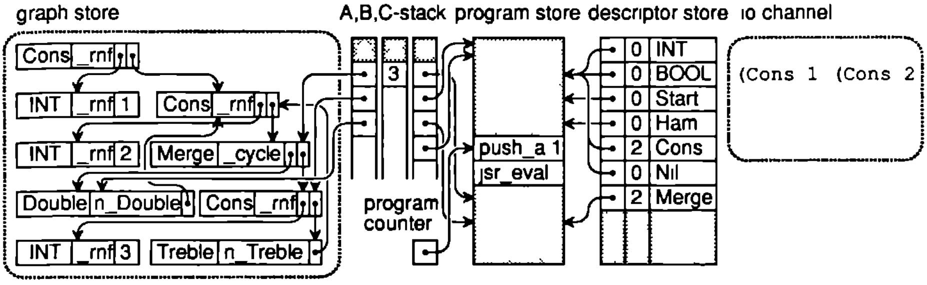


Fig 3.2 The ABC-machine

The corresponding state of the ABC-machine is defined by the following tuple:

```
state == (astack, bstack, cstack, graphstore, descstore, instrid, programstore, io)
         || This state is denoted in instructions as: (as,bs,cs,gs,ds,pc,ps,io)
```

The next section contains a thorough specification of these components.

3.2 The micro-instruction set

The components of the ABC-machine are abstract data-types. They are sufficiently specified by the operations defined on them. The Miranda implementation of the micro-instructions is not presented here, but it can be found in appendix A. These Miranda definitions specify the semantics of the micro-instructions, on the other hand they suggest too much a specific implementation. The given implementation must be viewed as a definition of the semantics, not as a suggestion for an efficient implementation. It is merely needed for detailed information like the index of the top of a stack.

The following type synonyms are used to increase the clarity of the type definitions. The type nat stands for natural numbers including zero, it is represented by Miranda numbers (num).

```
arity     == nat
a_dst     == nat    || the index of the destination on the A-stack
a_src     == nat    || the index of the source on the A-stack
b_dst     == nat    || the index of the destination on the B-stack
b_src     == nat    || the index of the source on the B-stack
c_src     == nat    || the index of the source on the C-stack
nr_args   == nat    || the number of arguments involved
arg_nr    == nat    || the number of the argument involved
```

To enhance the value of the specification in appendix A as prototype implementation, a number of additional rule alternatives are used to create sensible error messages.

The graph store

The graph in the *graph store* is basically a Clean-like graph. It is composed of *nodes*. Each node is labelled with an unique identification, a **node-id**.

There are micro-instructions to generate a new (empty) graph store, to create a new (empty) node in the graph and to get a node out of the graph. Finally, a node can be updated by a function passed as parameter to the gs_update micro-instruction.

```
gs_get      : nodeid -> graphstore -> node
gs_init     : graphstore
gs_newnode  : graphstore -> (graphstore,nodeid)
gs_update   : nodeid -> (node -> node) -> graphstore -> graphstore
```

Note that each of these micro-instruction names starts with gs_ (for graph store).

The nodes

Each ABC-node contains a **desc-id** (*descriptor identification*) to represent the symbol in the Clean-node. A desc-id is an entry in the descriptor store indicating the symbol, it is represented by the type descid. Furthermore, a node contains a sequence of node-id's representing the arguments. A node can also contain a basic value like an *integer* or *boolean*. To reduce the number of definitions only these two basic types are considered, so *characters*, *reals* and *strings* are not treated here.

The graph in the graph store will be examined by the program which will try to reduce the graph to normal form. For this purpose it is convenient that additional information is stored in the nodes of the graph. A node contains a *context*; the **instr-id** (instruction identification) referring to the first instruction of an instruction sequence. By convention, this ABC-instruction sequence will reduce the corresponding node to root normal form when it is executed. The instr-id stored in a node can be changed during reduction. This is used for several markings of the node. For instance, to determine at run-time that a node is already under reduction, in this way cyclic computations can be detected.

There are micro-instructions to extract the pieces of information stored in a node. The nodes can be extracted of the graph store by the micro-instruction gs_get defined above.

```
n_arg       : node -> arg_nr -> arity -> nodeid
n_args      : node -> arity -> nodeid_seq
n_arity     : node -> arity
n_B         : node -> boolean
n_descid    : node -> descid
```

```
n_entry      :: node -> instrid
n_I          :  node -> int
n_nargs      :  node -> nr_args -> arity -> nodeid_seq
```

There are also micro-instructions to test whether a piece of a node has a certain value. These micro-instructions are important for pattern matching.

```
n_eq_arity   :  node -> arity -> boolean
n_eq_B       :  node -> boolean -> boolean
n_eq_descid  :  node -> descid -> boolean
n_eq_I       :  node -> int -> boolean
n_eq_symbol  :: node -> node > boolean
```

Finally, the contents of a node can be changed by:

```
n_copy       :  node -> node -> node
n_fill       :  descid -> instrid -> args -> node -> node
n_fillB      :  descid -> instrid -> boolean -> node -> node
n_fillI      :: descid -> instrid -> int -> node -> node
n_setentry   :  instrid -> node -> node
```

The micro-instructions to change a node are passed as argument to gs_update to overwrite nodes in the graph.

The micro-instructions above show that the graph store deals with variable sized nodes. Overwriting a previously created node with new values is always possible. In general, one first has to create a new empty node which has to be filled later. This method is in particular convenient for the creation of cyclic graphs. The code generation schemes presented in the chapter 5 rely on the presence of a node that can be overwritten with the result of a reduction.

The descriptor store

The ABC-machine contains a piece of memory where symbol descriptors are stored. This *descriptor store* contains information about the symbols used in the rewrite system. Given the descriptor identification, descid, a descriptor can be taken from the descriptor store. This store can be initiated by passing a list of descriptors to the ds_init micro-instruction.

```
ds_get       :  descid -> descstore -> desc
ds_init      :  [desc] -> descstore
```

The descriptors

A *descriptor* contains information of the associated symbol; its arity, the start address of the reduction code and the name of the symbol. The symbol name is only used to print a representation of the graph in root normal form on the output channel. Information can be retrieved from the descriptor by the corresponding micro-instructions.

```
d_ap_entry   :  desc -> instrid
d_arity      :  desc -> arity
d_name       :  desc -> string
```

The A-stack

The *A-stack* contains node-id's; references to nodes stored in the graph store. It is used to access the actual arguments and the result of the rewrite rule executed. Just as in other imperative languages, not a complex data structure is passed to or returned from a function, but a reference to these objects is passed. The top of the stack has index 0 (as is the case for the other stacks).

A new, empty, A-stack can be created by:

```
as_init      :  astack
```

An element at any depth, or a sequence of nr_args top elements, can be retrieved from the A-stack by:

```
as_get       :  a_src -> astack -> nodeid
as_topn      :  nr_args -> astack -> nodeid_seq
```

The type nodeid_seq represents a sequence of node-id's which can be taken from or pushed on the A-stack, it serves also as the argument sequence for a node. The A-stack can be updated by:

```
as_popn      :  nr_args -> astack -> astack
as_push      :  nodeid -> astack -> astack
as_pushn     :  nodeid_seq -> astack -> astack
as_update    :  a_dst -> nodeid -> astack -> astack
```

The B-stack

If the calculation of a simple numerical value would be performed by building nodes, performing redirections and the like, it is obvious that in no way an efficient implementation can be achieved on traditional hardware which has no support for these kind of actions whatsoever. On a traditional machine simple calculations are performed on a stack using only a couple of simple instructions. In order to obtain the desired behaviour for these calculations, the ABC-machine is equipped with a stack to hold basic values.

A further optimization is the use of registers to hold these basic values. However, registers are not included in the ABC-machine design. The number of registers and the operations possible on them varies too much between different concrete machines. Register allocation is left as a part of the implementation of the ABC-machine.

The *B-stack* contains basic values like integers and booleans. Such values are stored untagged on the B-stack. This means that it is impossible to determine the type of the elements of the B-stack.

The initial, empty, B-stack is created by:

```
bs_init     :: bstack
```

Information can be obtained from the B-stack with the micro-instructions:

```
bs_get      :: b_src -> bstack -> basic
bs_getB     :: b_src -> bstack -> boolean
bs_getI     :: b_src -> bstack -> int
```

The B-stack can be changed with the micro-instructions:

```
bs_popn     :: b_src -> bstack -> bstack
bs_push     :: basic -> bstack -> bstack
bs_pushB    :: boolean -> bstack -> bstack
bs_pushI    :: int -> bstack -> bstack
bs_pushn    :: basic_seq -> bstack -> bstack
bs_update   :: b_dst -> basic -> bstack -> bstack
```

Beside these updating micro-instructions there are micro-instructions defined to perform computations with the basic values stored on the B-stack. Usually the arguments are all on top of the B-stack and are replaced by the result of the operation. When arguments are not on the B-stack they are an argument of the micro-instruction. Some micro-instructions to handle integer values are:

```
bs_addI     :: bstack -> bstack
bs_eqI      :: bstack -> bstack
bs_eqIi     :: int -> b_src -> bstack -> bstack
bs_gtI      :: bstack -> bstack
```

The program

The ABC-machine contains a sequence of machine instructions representing the reduction algorithm: the *program*. This program will rewrite the initial graph to its normal form according to the annotated functional strategy. Conceptually there are two algorithms involved, the annotated functional reduction strategy which indicates the next redex and the rewriting of that redex according the Clean rules. These algorithms are merged in the ABC-program to increase the efficiency. Each Clean rule is translated into a sequence of ABC-instructions. This instruction sequence controls the order of reductions required for this rule and performs the reduction according to the first matching rule alternative.

Each instruction has an unique identification, the *instr-id*, in order to be indicated as the current instruction by the program counter. A program does not change during execution, it is loaded in the machine when the machine is booted. Programs to be stored are denoted by a list of instructions.

```
ps_get     :  instrid -> programstore -> instruction
ps_init    :  [instruction] -> programstore
```

The program counter

Since the ABC-machine has an imperative nature it is essential to have a locus of control; a *program counter*. The program counter contains the instr-id of the next instruction to be executed.

There are micro-instructions provided to initiate the program counter (points to the first instruction of the program) and to change it. The program counter can be incremented (i.e. the next instruction becomes the current instruction), or can be set outside the program to indicate that the program is finished. Finally, one can check whether the last instruction of the program is reached.

```
pc_init       :  instrid
pc_next       :  instrid -> instrid
pc_halt       :  instrid -> instrid
pc_end        :  instrid -> bool
```

The C-stack

The *C-stack* (Control stack) is used to implement nested reductions in the abstract machine. The program counter can be stored and recovered from this stack.

```
cs_init       :  cstack
cs_get        :  c_src -> cstack -> instrid
cs_popn       :  nr_args -> cstack -> cstack
cs_push       :  instrid -> cstack -> cstack
```

The input-output channel

The abstract machine furthermore contains an *input output channel* used to show the result of the reduction to the world outside. On the output channel strings can be printed. These strings are appended to the existing output channel.

```
io_init    :  io
io_print   :  string -> io -> io
```

3.3 The instruction set

Each machine instruction of the ABC-machine is defined in terms of the micro-instructions. Each instruction consists of an instruction identifier and zero or more operandi. Not all instructions are shown in this section. The specification in terms of micro-instructions is only given for the most important instructions. For some other instructions only the type and an informal explanation is given. A precise definition of all machine instructions is given in appendix A.

The instructions shown here are classified according to their main purpose:

- graph manipulation;
- retrieving information from a node;
- manipulating the A-stack;
- manipulating the B-stack;
- changing the flow of control;
- generating output.

Graph manipulation

There are several kinds of instructions to manipulate the graph store. There is only one instruction to create a new node in the graph store. Several instructions can be used to change the contents of existing nodes. Finally, there are also instructions that fetch information stored in the nodes of the graph. All instructions for graph manipulation (with exception of the instruction create) have as operand an offset in the A-stack, to find the node-id of the node to manipulate.

The instruction create creates a new empty node in the graph store, the node-id of the new node is pushed on the A-stack. It is defined as:

```
create (as,bs,cs,gs,ds,pc,ps,io)
 = (as',bs,cs,gs',ds,pc,ps,io)
   where  as'            = as_push nodeid as
          (gs', nodeid)  = gs_newnode gs
```

create is the only instruction to create a new (empty) node. All other instructions can only change the contents of an already existing node. The most elaborated instruction to update the contents of a node is fill, which is defined as:

```
fill desc nr_args instrid a_dst (as,bs,cs,gs,ds,pc,ps,io)
 = (as',bs,cs,gs',ds,pc,ps,io)
   where  as'     = as_popn nr_args as
          gs'     = gs_update nodeid (n_fill desc instrid args) gs
          nodeid  = as_get a_dst as
          args    = as_topn nr_args as
```

As can be seen from this specification the arguments of the node are taken from the A-stack. The first argument is on top of the A-stack.

Other instructions to change the contents of an existing node are:

```
fill_a     :: a_src -> a_dst -> instruction   || the copy node instruction;
fillB      :: bool -> a_dst -> instruction    || fills the node with the given boolean;
fillB_b    :: b_src -> a_dst -> instruction   || fills with a boolean found at the B-stack;
fillI      :: int -> a_dst -> instruction     || fills the node with the given integer;
fillI_b    :: b_src -> a_dst -> instruction   || fills with the integer found at the B-stack
set_entry  :: instrid -> a_dst -> instruction || changes the reduction context of the node.
```

Building a Clean-node in the graph store involves at least two instructions: a create, which leaves a node-id on the A-stack and one of the fill instructions.

Example: to create the graph Cons 1 Nil, the following instruction sequence can be used. Assume that "_rnf" indicates some ABC-instruction sequence, probably just containing a rtn instruction since the newly created nodes are already in root normal form. The descriptor-id's of Nil and Cons are indicated by "Nil" and "Cons". This code fragment is written in the ABC-assembly language introduced below.

```
[       Create                        ,  || node for Cons
        Create                        ,  || node for Nil; 2nd arg of Cons
        Fill     "Nil" 0 "_rnf" 0     ,  || fill node just created
        Create                        ,  || node for 1; 1st arg of Cons
        FillI    1 0                  ,  || fill node just created
        Fill     "Cons" 2 "_rnf" 2    ]  || fill Cons node
```

Retrieving information from a node

The retrieved information is stored on one of the stacks or in the program counter. The node-id of the node is found at the indicated depth on the A-stack. There are also instructions to test whether the contents of a node has the given value.

```
push_args a_src arity nr_args (as,bs,cs,gs,ds,pc,ps,io)
 = (as',bs,cs,gs,ds,pc,ps,io)
   where as'    = as_pushn args as
         args   = n_nargs (gs_get nodeid gs) nr_args arity
         nodeid = as_get a_src as

pushI_a a_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where bs'    = bs_pushI int bs
         int    = n_I (gs_get nodeid gs)
         nodeid = as_get a_src as
```

```
eqI_a int a_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'     = bs_pushB equal bs
          equal   = n_eq_I (gs_get nodeid gs) int
          nodeid  = as_get a_src as

eq_desc_arity descid arity a_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'     = bs_pushB equal bs
          equal   = (n_eq_descid node descid) & (n_eq_arity node arity)
          node    = gs_get nodeid gs
          nodeid  = as_get a_src as
```

In appendix A some other instructions to extract (a selection of) the arguments from a node, change the arguments or to test on other basic value types are given.

Manipulating the A-stack

The A-stack is used to access the nodes involved in a rewriting. Via push instructions new node-id's can be pushed on the stack. In addition the following instructions are provided to manipulate the A-stack:

```
pop_a nr_args (as,bs,cs,gs,ds,pc,ps,io)
 = (as',bs,cs,gs,ds,pc,ps,io)
   where  as'     = as_popn nr_args as

push_a a_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as',bs,cs,gs,ds,pc,ps,io)
   where  as'     = as_push nodeid as
          nodeid  = as_get a_src as

update_a a_src a_dst (as,bs,cs,gs,ds,pc,ps,io)
 = (as',bs,cs,gs,ds,pc,ps,io)
   where  as'     = as_update a_dst nodeid as
          nodeid  = as_get a_src as
```

Example: the cyclic graph ones: Cons 1 ones, is constructed similar to the previous example. Here the advantages of a separate create and fill instruction are visible.

```
[         Create                        ,  || node for Cons
          Push_a   0                    ,  || 2nd arg of Cons
          Create                        ,  || node for 1; 1st arg of Cons
          FillI    1 0                  ,  || fill node just created
          Fill     "Cons" 2 "_rnf" 2    ]  || fill Cons node
```

Manipulating the B-stack

For the B-stack the same stack handling instructions as for the A-stack are defined:

```
pop_b     :: nr_args -> instruction
push_b    :: b_src -> instruction
update_b  :: b_src -> b_dst -> instruction

pushI     :: int -> instruction
pushB     :: bool -> instruction
```

There are also instructions to manipulate the basic values on this stack. A typical example is

```
addI (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'      =  bs_addI bs
```

There are many more instructions to do arithmetic, they all follow the same scheme as the add instruction presented here. They are not listed here for reasons of brevity.

Changing the flow of control

The desired flow of control has to be realized by manipulating the program counter. Jumps are unconditional, or directed by the boolean value on top of the B-stack.

```
jmp address (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs,gs,ds,pc',ps,io)
   where  pc'      =  address

jmp_false address (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc',ps,io)
   where  pc'      =  cond (bs_getB 0 bs) pc address
          bs'      =  bs_popn 1 bs

jmp_true address (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc',ps,io)
   where  pc'      =  cond (bs_getB 0 bs) address pc
          bs'      =  bs_popn 1 bs
```

When a jsr (jump to subroutine) instruction is executed, the current value of the program counter is stored on the C-stack, a rtn (return from subroutine) instruction will restore the program counter and pop the return address from the C-stack.

```
jsr address (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs',gs,ds,pc',ps,io)
   where pc'     = address
         cs'     = cs_push pc cs

rtn (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs',gs,ds,pc',ps,io)
   where pc'     = cs_get 0 cs
         cs'     = cs_popn 1 cs
```

A jsr_eval instruction will start the execution of the code addressed in the node referenced by the top of the A-stack, the return address is saved on the C-stack. Hence, it performs a jsr to the address stored in the node. By convention, executing the instruction sequence will reduce the node to its root normal form.

```
jsr_eval (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs',gs,ds,pc',ps,io)
   where pc'     = n_entry (gs_get nodeid gs)
         nodeid  = as_get 0 as
         cs'     = cs_push pc cs
```

The program execution stops after the execution of a halt instruction. The instruction fetch_cycle (see below) will be left when the program counter contains an address outside the program.

```
halt (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs,gs,ds,pc',ps,io)
   where pc'     = pc_halt pc
```

Generating output

To show the result of the reduction there are print instructions. These instructions append strings to the output channel.

```
print string (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs,gs,ds,pc,ps,io')
   where io'     = io_print string io

print_symbol a_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs,gs,ds,pc,ps,io')
   where io'     = io_print string io
         node    = gs_get (as_get a_src as) gs
         string  = symbol_to_string node desc
         desc    = ds_get (n_descid node) d
```

3.4 Program execution

To run the ABC-machine described here, the instructions must be applied to the state of the machine. Once started the machine must continue to execute instructions until the halt instruction is executed. While the machine is running, the current instruction is fetched continuously out the program store and applied to the present state.

The instruction cycle

The fetch cycle recursively fetches the current instruction out the program and applies it on the current state. The fetching (and hence the machine) stops when the program counter indicates that a halt instruction is executed.

```
fetch_cycle :: state -> state
fetch_cycle (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs,gs,ds,pc,ps,io), pc_end pc
 = fetch_cycle (currinstr (as,bs,cs,gs,ds,pc',ps,io)), otherwise
   where pc'       = pc_next pc
         currinstr = ps_get pc ps
```

Booting the machine

Before it is possible to run the machine, the machine is loaded (booted) with the initial state. The program and descriptors must be supplied as argument to the boot function. All parts of the machine are initiated by the corresponding init micro-instructions. The machine starts evaluating the program on the first instruction.

```
boot :: ([instruction],[desc]) -> state
boot (program,descriptors)
 = (as,bs,cs,gs,ds,pc,ps,io)
   where pc  = pc_init
         as  = as_init
         bs  = bs_init
         cs  = cs_init
         gs  = gs_init
         ps  = ps_init program
         io  = io_init
         ds  = ds_init descriptors
```

3.5 ABC-assembler

In order to write ABC-programs there must be a denotation for all objects involved. The denotation of the function arguments is specified here.

```
boolean    ==  bool     || A boolean is represented by the Miranda type bool.
context    ==  instrid  || A context is represented by the address of the first instruction.
descid     ==  instrid  || A descriptor-id is the address of the descriptor in the store.
instrid    ==  nat      || An address is the number of the instruction in the sequence.
int        ==  num      || Integers are represented by Miranda num's
nat        ==  int      || Natural numbers are also represented by Miranda num's.
```

References to instructions are made by the address of the instruction within the program. This is fine for the ABC-machine and close to reality in a concrete machine, but cumbersome to read and write for human beings. Moreover the standard Miranda system does not support functions on top level very well. The system just prints <function>, which is not a clear identification of a specific ABC-instruction. To make ABC-programs better understandable an assembler level is introduced. Assembler statements can be mapped directly to instructions, but the assembler uses labels instead of addresses. A program in ABC-assembler is represented by a Miranda data structure.

```
assembler   ==  [statement]
label       ==  [char]
desc_label  ==  label          || the label of a descriptor
red_label   ==  label          || the label of an instruction

statement
            ::= Label       label                                 |
                Descriptor  desc_label red_label arity name       |
                Create                                            |
                Fill        label nr_args label a_dst             |
                Jmp         label                                 |
                etc...
```

Every ABC-instruction is represented by a constructor with a similar name, the first character is changed in an uppercase character to make it a constructor. Generally also the arguments are identical, only addresses are replaced by labels. ABC-assembler also contains labels and descriptor definitions. See appendix A for a complete definition of ABC-assembler statements and an assembler.

Example: this assembler form to represent ABC-instructions allows us to write:

```
example :: assembler
example
= [          Label
  "Length1"  Jsr_eval                                   ,
             Eq_desc_arity   "Cons" 0 0                 ,
             Jmp_false       "Length2"                  ,
             Push_args       0 2                        ,
             Create                                     ,
             Push_a          2                          ,
             Fill            "Length" 1 "n_Length" 1    ]
```

Instead of the corresponding ABC-instruction sequence:

```
example :: program
example
= [          jsr_eval                                   ,
             eq_desc_arity   23 0 0                     ,
             jmp_false       62                         ,
             push_args       0 2                        ,
             create                                     ,
             push_a          2                          ,
             fill            12 1 613 1                 ]
```

The reason for introducing a separate assembler level is not only to enhance the readability of the program, but also to simplify the translation of Clean rules to ABC-code. This simplification is caused by the use of symbolic names and labels instead of the corresponding numbers of the ABC-machine. To have ABC-assembler as a Miranda data-structure has also some advantages for the prototype; it eliminates the generation and parsing of ABC-assembler represented as a list of characters and enables the printing and manipulating of statements which is impossible for a sequence of instructions.

3.6 Comparison with the G-machine

Another well known abstract machine for graph reduction is the G-machine [Johnson 84]. The objectives of this machine are identical to that of the ABC-machine: defining an imperative abstract graph rewrite machine as an intermediate level in the compilation of functional languages. The G-machine is not described in a functional language, and is initially meant for graphs in the applicative form (each argument is bound to the function by an AP node). Due to these two reasons both the description and the described abstract machines differ in almost all details. An overall view of the ABC-machine and the G-machine shows many correspondences.

In order to compare the G-machine and its description with the ABC-machine, the G-machine state and some instructions in the G-machine formalism are shown. Afterwards the main differences between description methods and abstract machines are indicated. The description method used for the G-machine very much resembles the description method used by Landin to describe the SECD-machine [Landin 64].

The G-machine state

A state in the abstract G-machine is a 7-tuple <O,C,S,V,G,E,D>, where

O is the *output* produced so far.

C is the G-*code* sequence currently being executed (sometimes called *code-stack*).

S is a *stack* of node names i.e. pointers into the graph.

V is a stack of basic *values*.

G is the *graph*. A mapping from node-names to nodes. There are nodes of the following type:

INT i integer nodes,

BOOL b boolean nodes,

NIL empty-list nodes,

CONS n_1 n_2 list nodes, where n_1 is the pointer to the head graph and n_2 is the pointer to the tail graph,

AP n_1 n_2 application nodes, where n_1 is a pointer to the function graph and n_2 is a pointer to the argument graph,

FUN f a node with a reference to the compiled function f,

HOLE a node which is to be filled with another value later; it is used during the construction of cyclic graphs.

E is the global *environment*, which is a mapping from function names to pairs consisting of the number of curried arguments of the function and its code sequence.

D is a *dump* used for recursive calls to EVAL: a stack of pairs consisting of; a stack of node names: S before EVAL, and a G-code sequence: C before EVAL.

Comparison of machine states.

The correspondence of the machine states is shown by indicating which components of the ABC-machine correspond to the elements of the 7-tuple of the G-machine described above.

O The output has a direct correspondence with the output in the ABC-machine.

C In the G-machine the current instruction is the head of the code sequence C. The flow of control involves the movement of code between the code sequence C, the environment E, and the dump D. In the ABC-machine there is no movement of code; the instruction sequence executed is controlled by the program counter (pc) and the control stack (C-stack).

S The stack, S, of the G-machine directly corresponds to the argument stack (A-stack) of the ABC-machine.

V The stack of basic values, V, corresponds to the basic value stack (B-stack), although it is used in a more restricted way in the G-machine.

G The graphs of both machines obviously correspond. The G-machine contains only binairy nodes while the nodes in the ABC-machine are variable sized. Moreover, the nodes of the ABC-machine contain an extra field for the context and information about the constructor can be found via the descriptor-id identifying the symbol of the node. The type list is the only data structure in the G-machine, it is "wired" in the definition. In the ABC-machine there are no assumptions on the data types used, the corresponding constructors are sufficiently specified by their descriptors in the descriptor store.

E The environment, E, of the G-machine contains the arity and the code corresponding to the functions. The arity is stored in the descriptor store of the ABC-machine, and the code in the program.

D The dump of the G-machine is not found in the ABC-machine. The stack is not stored separately on an other stack in the ABC-machine, the actions needed (number of pops upon returning from a function) are computed during the compilation of Clean to ABC-code. The code sequences stored in the dump of the G-machine, are recorded by the program counters stored on the C-stack of the ABC-machine.

The G-machine instructions

In a G-machine state, () denotes an empty stack or an empty code sequence. The initial graph is indicated by {}. The semi-colon appends values onto an output sequence. The period is used as infix cons for instruction sequences and push for stacks. Updating of the graph is written as G[n=INT i]. If there is a node named n previously in G, then the node n is updated with a new value, otherwise a new node is created. This notation is also used in pattern matching situations.

The flow of control is conducted by the following instructions. Descriptions are taken from [Johnson 84], with a few errors corrected (see also [Peyton Jones 87]).

<o, EVAL.c, n.s, v, G[n=AP n_1 n_2], E, D>
  => <0, UNWIND.(), n.(), v, G[n=AP n_1 n_2], E, (c,s).D>
<o, EVAL.c, n.s, v, G[n=INT i], E, D>
  => <0, c, n.s, v, G[n=INT i], E, D>,
  similarly for nodes BOOL b, Nil, CONS n_1 n_2 and FUN f.

```
<o, UNWIND.(), n.s, v, G[n=AP n1 n2], E, D>
     => <o, UNWIND.(), n1.n.s, v, G[n=AP n1 n2], E, D>
<o, UNWIND.(), n0.n1...nk.s,v, G[n0=FUN f,n1=AP n0 n1',...,nk=AP n(k-1) nk'],E[f=(k,c)],D>
     => <o, c, n1'...nk'.s, v, G[n0=FUN f,n1=AP n0 n1',...,nk=AP n(k-1) nk'], E[f=(k,c)], D>
<o, UNWIND.(), n0.n1...nk.(),v, G[n0=FUN f],E[f=(a,c)],(c',s').D> and k<a
     => <o, c', nk.s', v,  G[n0=FUN f], E[f=(a,c)], D>

<o, RET m.c, n1...nm.n.(), v, G[n=INT i], E, (c',s').D>
     => <o, c', n.s', v, G[n=INT i], E, D>,
     similarly for nodes BOOL b, NIl and CONS n1 n2
<o, RET m.c, n1...nm.n.s, v, G[n=AP n1 n2], E, D>
     => <o, UNWIND.(), n.s, v, G[n=AP n1 n2], E, D>,
     similarly for n = FUN f

<o, JFALSE l.c, s, true .v, G, E, D>    => <o, c, s, G, E, D>
<o, JFALSE l.c, s, false.v, G, E, D>    => <o, JMP l.c, s, G, E, D>

<o, JMP l...LABEL l.c, s, G, E, D>      => <o, c, s, G, E, D>

<o, LABEL l.c, s, G, E, D>              => <o, c, s, G, E, D>
```

Initial and final state of the G-machine

The initial configuration of the machine for the evaluation of the expression e_0 in environment E_0 is:

```
< (), c0, () , (), {}, E0, ()>
where c0 =    E[e0]r00;PRINT
```

This is a machine with an empty output, the code for evaluating the start expression ($E[e_0]r_0 0$), an empty basic value stack, an empty graph, environment E_0 containing the compiled code for the functions together with their arity, and an empty dump.

The machine stops when the state <o, (), (), (), G, E, ()> has been reached.

Differences between the abstract machines

Apart from the way in which the machines are described, they also differ as abstract graph rewrite machines. The most important distinctions are:

- The nodes in the graphs of both machines are different. The nodes in the ABC-machine contain an independent context and descriptor-id, whereas the nodes in the G-machine only hold a type identifier.
- The kind of graphs handled by the G-machine is different from the graphs treated in the ABC-machine. In the ABC-machine nodes have variable arity, while the nodes in the G-machine are either leave nodes or AP nodes with two arguments.

The graph Ackerman 2 3 is represented in the G-machine as AP (AP (FUN Ackerman) (INT 2)) (INT 3). Later on, variants of the G-machine are developed which use a more efficient representation for these graphs. [Burn 88, Johnson 87]

- The type list is the only data type available in the G-machine; special instructions are available to handle it. In the ABC-machine arbitrary constructors are handled.

As shown in the next chapter, the generated code for the machines also differs, e.g. the calling conventions of functions are different, and the B-stack is used in a more general way than the stack V.

Differences between the descriptions

The G-machine is specified in another formalism than the ABC-machine, although the tuple describing the machine state shows some resemblance. The main differences between the descriptions are:

- The G-machine is not specified in a programming language; this implies that it can not be checked by a compiler, nor serve as a prototype implementation. The original specification of Johnson contains some errors (like unbound variables in the right hand side of a description rule), that would be spotted by a compiler.
- The G-machine description is not hierarchical, this implies that all items must be addressed at the same level. In the ABC-machine implementation details can be hidden in the micro-instructions.
- The instruction sequence executed is specified in the ABC-machine in a more direct way than in the G-machine. The flow of control is also more restricted in the G-machine, e.g. jumps are only forward and within the code resembling to one function. In later variants of the G-machine a more elaborated flow of control is defined [Johnson 87].
- The G-machine description is not complete, some cases (see EVAL and RET instructions) are specified by similarly for nodes.... This is not always perfectly clear (e.g. must the EVAL instruction for the node FUN f behave similar to the first rule, or to the second rule of the given specification).
- The described G-machine is essentially an interpreter, whereas the ABC-machine contains directly executable code. The instructions of the ABC-machine are functions that transform the machine state. The state transformations of the G-machine must be executed by some external function.

3.7 Discussion

This chapter presents a large and complex example of the description method introduced in the previous chapter. The abstract machine specification presented is actually *the* specification of the ABC-machine. It was not made after the completion of the design, but the machine was actually designed by writing this specification. The description is used with success as prototype implementation and it appeared to be a useful definition for the actual implementation. It appeared to be easy to add some instructions and to experiment with slightly different instruction sets. The development of this specification was forcing a well structured design and the development time was spent on designing the machine instead of solving problems with its specification.

The ABC-machine is an imperative graph rewrite machine, it is an abstraction of concrete stack based machine architectures. Its instruction set provides an additional level of abstraction for the implementation of functional languages. The introduction of this level enhances portability, without a significant reduction in efficiency. The assembly level provided permits the use of labels and symbolic names. The execution speed of the prototype is sufficient for small test programs. The Miranda interpreter on a SUN 3/280 executes circa 10 instructions each second. The nfib-number for the ABC-code shown in section 5.6 is 1. However, the execution speed can increased almost an order of magnitude by omitting some of the code to give meaningful error messages in the micro-instructions.

There are several implementations of the ABC-machine. The specification of the abstract ABC-machine appeared to be a very useful and a valuable definition of the abstract machine. The implementation in C of a parallel ABC-interpreter [Nöcker 89] follows the specification presented here very closely. The machine state is not passed from instruction to instruction, but as a global accessible data structure. By implementing the components of the ABC-machine in C a micro-instruction level is defined. The instructions are implemented using these micro-instructions. A state-of-the-art speed is achieved by the translation of ABC-code to MC68000 code for a SUN3/280 [Weijers 90, van Groningen 90].

Chapter 4
Abstract Program Translation

This chapter shows that functional programming languages can be used to describe program transformations and translations. The well-known bracket abstraction algorithm is used to compare this description method with some conventional descriptions. In the next chapter an elaborated example of the use of abstract program translations is shown.

The descriptions are called **abstract program translations** since the language to be transformed is represented by an **abstract syntax tree** (AST). In a functional language a data type is used to hold the AST. The structure of an AST is close to the structure of the language to be transformed. A translation is easier to express using the AST than using the syntax since the syntactical representation is just a linear sequence of characters, while an AST is a tree with the desired structure. Handling the syntactical structure in the description has as advantage that the syntactical sugar can be used in the description, but this description method appears to be only suited for small problems.

These data structures are not hidden in an abstract data type like the data structures used to represent the machine components in the previous chapters, but they are manipulated at the top level of the description. The translation functions use pattern matching on the data structure and deliver fragments of this data structure. Here the data structure is the representation of the objects manipulated, there is no reason at all to hide them in an abstract data structure. In the previous chapters the data structures themselves were irrelevant, just a carrier for the data was needed.

Efficiency and space consumption during the execution of the description is irrelevant. This implies that a simple data structure in close correspondence with the language to be transformed can be used. The data structures used in the description of program transformation do not resemble the data structure that would be used in a compiler. For a compiler efficiency and compactness of the data structures used are important constraints. This implies for instance the use of symbol tables.

In conventional descriptions of translations the language components handled are often surrounded by ⟦ and ⟧ to distinguish them from the translation algorithm. In practice however, it appears to be impossible or unpractical to make a strict separation between translation algorithm and translated language.

The abstract program translation methodology will be explained in section 4.1. Then several descriptions of bracket abstraction will be given in order to make a comparison of the description methods possible. The specifications shown here are the original description (in section 4.3), the specification using the syntactical match (in 4.4) and the abstract program translation (4.5). In the next chapter an abstract program translation is used to describe the translation of Clean to ABC-code. In appendix A it is used to describe the mappping from ABC-assembler code to instructions. It is also used by the author to specify the translation from Miranda to Clean [Koopman 88].

4.1 The description method

In order to specify a translation a suited representation of the languages involved must be available. Here the languages involved are represented by a data type in a functional language. The defined data type to represent the syntax tree is called an abstract syntax tree (AST). The translation is described by a function that takes the AST to transform as argument and delivers the resulting AST.

A suited type must be defined for each language involved in the translation. The choice of a good definition of this type appears to be crucial to obtain an elegant specification. The type definition cannot be obtained in some mechanical way. Valuable guide-lines for this definition are: keep it small, simple and close to the syntax tree. A direct translation of the syntax to a data type is often a good starting point, redundant constructs (like infix notation) are removed unless it is essential to describe their conversion to function applications. It is generally more convenient to express additional information as annotations in the AST than to derive this information when it is needed. Fields to insert these annotations are added to the type as required.

An AST can be made almost generally applicable by inserting a variable field for the name of a node and a list of arguments. In this way a very general applicable expression tree is defined.

```
tree   ::=  Node name args
args   ==   [tree]
name   ==   [char]
```

Although convenient in some circumstances the general use of this kind of tree is not advocated. Due to the lack of distinguishing structure the type system does not help to check the construction of consistent trees. We advocate to use tailor made trees as shown below.

As first example, a suited AST for the λ-calculus [Church 32, Barendregt 84] is shown. The λ-calculus handles λ-terms; objects with an extremely simple structure. The syntax of λ-terms is

```
term :=   '(' term term ')'  |  an application
          'λ' var '.' term   |  an abstraction
          var                   a variable
```

A data structure suited to represent λ-terms as AST in Miranda is

```
term ::=  AP term term   |  || an application
          ABS var term   |  || an abstraction
          VAR var           || a variable

var   ==  [char]            || a possible representation of variables
```

Example: The λ-term λ f.(λ g.(λ x.((f x) (g x)))) is represented by the data structure

```
ABS "f" (ABS "g" (ABS "x" (AP (AP (VAR "f") (VAR "x")) (AP (VAR "g") (VAR "x")))))
```

A variable occurs **free** in a term if it is not in the scope of an abstraction of that variable; it is **bound** otherwise. Using the tree representation of λ-expressions a variable is bound if it is a sub-expression, a leaf in the tree, of an abstraction of the same variable. The last occurrence of an abstraction of that variable seen from the root of the tree is said to **bind** the occurrence of that variable.

```
free_vars :: term -> [var]
free_vars (AP t1 t2)        = mkset (free_vars t1 ++ free_vars t2)
free_vars (ABS var term)    = free_vars term -- [var]
free_vars (VAR var)         = [var]
```

mkset is the library function to remove duplicated elements from a list.

The replacement of a free variable by some term is called **substitution**. The process of (repeated) substitution is called **reduction**. An expression that can be reduced is called a **redex** (reducible expression). To keep the definition simple it is required that the expression substituted does not contain free variables that become erroneously bound through the substitution. The proper reduction of such terms would require α-conversion (see below), since our purpose is just to show the use of an AST to describe a language it is not treated here.

$$\begin{aligned} x\,[x := N] &\equiv N\,; \\ y\,[x := N] &\equiv y,\ \text{if } x \neq y\,; \\ (\lambda x.M)\,[x := N] &\equiv \lambda x.M\,; \\ (\lambda y.M)\,[x := N] &\equiv \lambda y.(M\,[x := N]),\ \text{if } x \neq y\,;\ \ y \text{ is not allowed to occur free in } N \\ (M_1\,M_2)\,[x := N] &\equiv (M_1\,[x := N])\,(M_2\,[x := N]); \end{aligned}$$

```
substitute :: term -> var -> term -> term
substitute (VAR y)      x n   =  n          , if x = y
substitute (VAR y)      x n   =  VAR y   , if x ~= y
substitute (ABS y m)    x n   =  ABS y m                        , if x = y
substitute (ABS y m)    x n   =  ABS y (substitute m x n)  , if ~member (free_vars n) y
substitute (AP m1 m2) x n   =  AP (substitute m1 x n) (substitute m2 x n)
```

A change of a bound variable to a fresh variable in a term is called α-conversion.

$$\lambda x.M = \lambda y.M[x := y]$$

```
alphaconversion :: var -> term -> term
alpha_conversion y (ABS x m) =  ABS y (substitute m x (VAR y))
```

β-reduction is defined by.

$$(\lambda x.N)\ M \rightarrow_{\beta} N\ [x := M]$$

```
reduction :: term -> term
reduction (AP (ABS x n) m) =  substitute n x m
```

This example shows that a data structure can be chosen in close correspondence with the syntactical structure. Some transformations are shown to be easily expressible.

4.2 Bracket abstraction

In order to compare the description method proposed here with some other description methods, the well-known bracket abstraction algorithm will be described in several formalisms. Bracket abstraction was introduced by Turner [Turner 79] as implementation methodology for functional languages. The theoretical background and soundness of abstraction can be found in combinatorial logic [Schönfinkel 24, Curry 58].

In order to use bracket abstraction as an implementation technique for functional programming languages, every function is treated as a higher order function. This implies that arguments are bound one by one in each function application. Bracket abstraction is a program transformation that removes all variables (arguments) from a function body. Combinators from a small and fixed set are inserted in the function body to distribute the arguments during program execution. The absence of variables eliminates the need to maintain environments to associate actual arguments to variables during the execution of the program.

The language to transform is SASL [Turner 76], a simple predecessor of the language Miranda.

Combinator reduction

An expression in SASL is reduced in two steps. First it is transformed to combinator code as shown below. Then the combinator expression is reduced by applying the reduction rules for these combinators. The reductions are carried out in the **normal order**; the leftmost redex in the expression is continuously rewritten. For some combinators it is necessary to evaluate one or more arguments before the rewrite rule can be applied.

4.3 The traditional description of bracket abstraction

The description given here is a recapitulation of the relevant parts of Turner's famous paper. Turner utilised an extended version of the description method used by Curry. In order to make this section self-contained no knowledge of Turner's article, nor of combinatorial logic is assumed.

The basic abstraction algorithm

To remove variables from the source text the definition

```
def f x     = E
```

is considered. E is an expression in which function application is the only operation. It contains constants, including curried versions of operators, and variables. To transform an expression to its curried form functions are replaced by their curried equivalents and infix operators are replaced by the corresponding curried prefix operators. The example of the factorial function in the source language and in its curried version illustrates this transformation.

```
def fac n   = n =   0 -> 1 ; n * fac (n - 1)
def fac n   = cond (eq 0 n) 1 (times n (fac (minus n 1)))
```

The variable x can be removed from the definition above by abstracting it from the expression. This is denoted by

```
def f       = [x] E
```

In this definition, [x] E denotes the result of the operation which removes all occurrences of x from E. It is pronounced 'abstract x from E'. The step taken above is correct when the law of abstraction holds. This law states that abstraction is the inverse of application.

```
([x] E) x    = E
```

A **combinator** is a function without free variables. To distribute the arguments in the expression the combinators **S**, **K** and **I** are defined by their reduction rules.

```
S f g x      = f x (g x)
K x y        = x
I x          = x
```

Function application is denoted by juxtaposition, it associates to the left. Redundant brackets are usually omitted. Indicating the associations in the first combinator definition explicitly, it is written as:

```
((S f) g) x  = (f x) (g x)
```

This implies that

```
def f x y    = E                must be read as
def (f x) y  = E                applying the abstraction rule yields
def f x      = [y] E            which is
def f        = [x] ([y] E)
```

The abstraction algorithm is best explained by viewing expressions as trees. The abstraction algorithm considers various cases of these trees:

- Abstraction of a variable over an application, a fork in the tree, yields a S-combinator to distribute the argument over both sub-trees. The abstraction algorithm is recursively applied to both sub-trees.
- The abstraction of a variable from the same variable yields an I-combinator; the argument is needed here in the tree.
- The abstraction of a variable from all other variables and constants yields a K-combinator; the argument is not needed in this leaf of the tree and it is thrown away.

The abstraction algorithm is specified as (E_1 and E_2 are arbitrary expressions)

$$\begin{array}{lll} [x]\,(E_1\ E_2) & \Rightarrow \mathbf{S}\ ([x]\,E_1)\ ([x]\,E_2) & \\ [x]\,x & \Rightarrow \mathbf{I} & \\ [x]\,y & \Rightarrow \mathbf{K}\ y & \text{, where } y \text{ is a constant or a variable other than } x. \end{array}$$

Proof:

Using the law of abstraction and the reduction rules for the combinators (the expression rewritten is underlined).

$$\begin{array}{lll}
(\,\underline{[x]\;x}\,)\,x & = & \underline{\mathbf{I}\;x} \;=\; x \\
(\,\underline{[x]\;y}\,)\,x & = & \underline{\mathbf{K}\;y\;x} \;=\; y \\
(\,\underline{[x]\;(E_1\;E_2)}\,)\,x & = & \underline{\mathbf{S}\;([x]\,E_1)\;([x]\,E_2)\;x}\;) \\
 & = & (([x]\,E_1)\,x)\;(([x]\,E_2)\,x) \\
 & = & (E_1\;E_2)\;\;\text{by induction to the structure of terms.}
\end{array}$$

□

The abstraction of the argument from the definition for the factorial function in curried form

```
def fac n   =  cond (eq 0 n) 1 (times n (fac (minus n 1)))
```

yields

```
def fac     =  S (S (S (K cond) (S (S (K eq) (K 0)) I) (K 1)
               (S (S (K times) I) (S (K fac) (S (S (K minus) I) (K 1))))
```

Improving the generated combinator code

The definitions given above form a complete algorithm to remove variables from an expression. However, the combinator expression obtained by this algorithm is rather long winded. This is due to the nature of the abstraction algorithm used; each argument is brought down to every leaf of the expression tree. When the argument is not needed in a leaf it is discarded there by a K-combinator. The size of the expression can be reduced by discarding arguments as soon as possible and directing them only to the branches of the expression tree where they are needed. This is done by introducing some additional combinators and applying improvement rules from the leaves to the top of the tree[1].

The combinators needed to improve the code are defined by the rules:

```
B f g x   =  f (g x)
C f g x   =  f x g
```

The rules to improve the code are:

1. The distribution (by the S-combinator) of an argument over two sub-trees is not needed when the argument is discarded right away in both sub-trees (by a K-combinator). The argument can be discarded immediately.

[1] It is of course possible and more efficient to incorporate these improvement rules in the abstraction algorithm. For the sake of clarity, abstraction and optimization of the expression by the improvement rules are treated as separate algorithms here. Moreover, the combined algorithm is hard to specify elegantly in this formalism.

2. The distribution of an argument over two sub-trees is not needed if it is discarded in the left sub-tree and becomes the right sub-tree by an **I**-combinator.
3. The distribution of an argument which is thrown away in the left branch is equivalent with sending the argument to the right tree by a **B**-combinator.
4. The same holds when the argument is not needed in the right sub-tree. The argument can be passed to the left branch by a **C**-combinator.

These optimization rules are expressed formally as

$\mathbf{S}\,(\mathbf{K}\,E_1)\,(\mathbf{K}\,E_2)$	$\Rightarrow \mathbf{K}\,(E_1\,E_2)$		|| improvement rule 1
$\mathbf{S}\,(\mathbf{K}\,E_1)\,\mathbf{I}$	$\Rightarrow E_1$		|| improvement rule 2
$\mathbf{S}\,(\mathbf{K}\,E_1)\,E_2$	$\Rightarrow \mathbf{B}\,E_1\,E_2$	, if no earlier rule applies	|| improvement rule 3
$\mathbf{S}\,E_1\quad(\mathbf{K}\,E_2)$	$\Rightarrow \mathbf{C}\,E_1\,E_2$	, if no earlier rule applies	|| improvement rule 4

The first rule states that $\mathbf{S}\,(\mathbf{K}\,E_1)\,(\mathbf{K}\,E_2))$ and $\mathbf{K}\,(E_1\,E_2)$ are equivalent expressions. Such a relation can be proven easily. An arbitrary argument, x, is supplied and the reduction rules for the combinators are used to transform the expression.

Proof:

- rule 1

$$\begin{array}{ll}
\underline{\mathbf{S}\ (\mathbf{K}\ E_1)\ (\mathbf{K}\ E_2)\ x} & = \{\text{rewrite rule for S}\} \\
\underline{(\mathbf{K}\ E_1\ x)}\ (\mathbf{K}\,E_2\,x) & = \{\text{rewrite rule for K}\} \\
E_1\ \underline{(\mathbf{K}\ E_2\ x)} & = \{\text{rewrite rule for K}\} \\
E_1\,E_2 &
\end{array}$$

and

$$\begin{array}{ll}
\underline{\mathbf{K}\ (E_1\ E_2)\ x} & = \{\text{rewrite rule for K}\} \\
E_1\,E_2 &
\end{array}$$

hence $\mathbf{S}\,(\mathbf{K}\,E_1)\,(\mathbf{K}\,E_2)$ and $\mathbf{K}\,(E_1\,E_2)$ are extensionally equal.

- rule 2

$$\underline{\mathbf{S}\ (\mathbf{K}\ E_1)\ \mathbf{I}\ x} \quad = \underline{(\ \mathbf{K}\ E_1\ x\ )}\,(\mathbf{I}\,x) \ = E_1\,(\,\underline{\mathbf{I}\ x}\,) \ = E_1\,x$$

hence $\mathbf{S}\,(\mathbf{K}\,E_1)\,\mathbf{I}$ and E_1 are extensionally equal.

- rule 3

$$\begin{array}{ll}
\underline{\mathbf{S}\ (\mathbf{K}\ E_1)\ E_2\ x} & = \underline{(\ \mathbf{K}\ E_1\ x\ )}\,(E_2\,x) = E_1\,(E_2\,x) \\
\underline{\mathbf{B}\ E_1\ E_2\ x} & = E_1\,(E_2\,x)
\end{array}$$

hence $\mathbf{S}\,(\mathbf{K}\,E_1)\,E_2$ and $\mathbf{B}\,E_1\,E_2$ are extensionally equal.

- rule 4

$$\begin{array}{ll}
\underline{\mathbf{S}\ E_1\ (\mathbf{K}\ E_2)\ x} & = E_1\,x\,\underline{(\ \mathbf{K}\ E_2\ x\ )} = \ E_1\,x\,E_2 \\
\underline{\mathbf{C}\ E_1\ E_2\ x} & = E_1\,x\,E_2
\end{array}$$

hence $\mathbf{S}\,E_1\,(\mathbf{K}\,E_2)$ and $\mathbf{C}\,E_1\,E_2$ are extensionally equal. □

This proof shows also that the improved expressions require less reduction steps; the expressions are indeed enhanced.

Using these rules the combinator code for the factorial function becomes:

```
def fac  =  S (C (B cond) (eq 0)) 1 (S times (B fac (C minus 1)))
```

These simple optimizations give a considerable reduction of the size of the expressions. Reductions of 50 % and more are common. The code can be further improved by defining additional combinators and considering more special cases [Turner 79, Peyton Jones 87], since our aim is not to describe the best possible fixed combinator generation this is not done.

Data structures

There is only one data structure defined in SASL; the list. Lists are built by the pairing operator and the empty list **nil**. The pairing operator (called cons) is represented by an infix colon ':'. In prefix form it is the combinator **P** (for 'pair'). The combinator **U** (for 'uncurry'[2]) unpacks pairs.

```
U f (P x y)  =  f x y
```

The left hand side of a function definition in SASL may be, or contain, a list. Hence the abstraction algorithm has to be extended with respect to abstractions of pairs. The Pair is unpacked and both elements are abstracted.

```
[P x y] E     ⇒ U ([x] ([y] E))
```

The use of this abstraction rule is illustrated by the example shown below.

```
def f (a:b)    =  a + b                      becomes in prefix form
def f (P a b)  =  plus a b                   abstracting the function argument
def f          =  [P a b] (plus a b)         which is after abstraction
def f          =  S (B plus (U K)) (U (K I))
```

Local definitions

In SASL, every expression can be accompanied by a sequence of locally defined functions. This is transformed to a single combinator expression in three steps.

1. The arguments of the locally defined functions are removed.

```
E1   where f x   =  E2
           g y   =  E3          is in the first step transformed to:

E1   where f     =  [x] E2
           g     =  [y] E3
```

[2] This name is due to Turner, 'unpair' or 'unpack' seems to be a better name.

2. The sequence of locally defined functions is grouped to a single local definition. This is done by packing them in a list.

   E_1 **where** f = E_2
   g = E_3 is in the second step transformed to:

   E_1 **where** (f:g) = (E_2:E_3)

3. Finally, the resulting local definition is removed. This is done by abstracting the list containing the locally defined function names from the expression E_1. The locally defined function body is supplied as the corresponding argument.

   E_1 **where** f = E_2 is transformed to:

   ([f] E_1) E_2

Recursion

There is one further complication to handle; when the locally defined functions are (mutual) recursive, the algorithm outlined above would not abstract the names of the locally defined functions from their body. This is illustrated by an example of the code produced by the algorithm above.

```
fac 3 where fac n  =  n=0 -> 1; n * fac (n - 1)

(C I 3) (S (C (B cond) (eq 0)) 1 (S times (B fac (C minus 1))))
```

The code produced is obviously wrong! It still contains the identifier fac. Recursion can be handled correctly by using the fixed point combinator Y. A term *e* is a **fixed point** of a function *f* when $f\ e = e$. The **fixed point combinator** is defined such that, for any function f, Y f is a fixed point of f.

```
Y f   = f (Y f)
```

A correct algorithm to handle local definitions removes the occurrence of the locally defined function name from its body. Then, this function is removed from the expression

E_1 **where** f = E_2 is first transformed to
E_1 **where** f = Y ([f] E_2) this is transformed to
([f] E_1) (Y ([f] E_2))

Using this transformation rule, the factorial example above becomes

```
(C I 3) (Y (B (S (C (B cond (eq 0))1)) (B (S times) (C (B S K) (C minus 1))))
```

Global function identifiers are not removed from the expression at compile time, but they are replaced by the associated combinator expression when they are encountered at run-time. So, recursive functions are no problem. Since this implementation is outside the world of fixed combinators discussed here, it is not treated in the sequel.

Summary of the abstraction algorithm

In order to make a comparison of the description methods, the rules of the abstraction algorithm are summarised. The equivalent of these rules in the other formalisms will be given below.

The combinators used in the abstraction process are defined by their reduction rules.

$$\begin{array}{lll}
\mathbf{S}\ f\,g\,x & = & f\,x\,(g\,x) \\
\mathbf{K}\ x\,y & = & x \\
\mathbf{I}\ x & = & x \\
\mathbf{B}\ f\,g\,x & = & f\,(g\,x) \\
\mathbf{C}\ f\,g\,x & = & f\,x\,g \\
\mathbf{U}\ f\,(\mathbf{P}\,x\,y) & = & f\,x\,y \\
\mathbf{Y}\ f & = & f\,(\mathbf{Y}\,f)
\end{array}$$

Arguments are removed from the definition (E is an expression in which function application is the only operation)

$$\mathbf{def}\ f\,x \quad = \quad E$$

by abstracting them from the body. This is denoted as

$$\mathbf{def}\ f \quad = \quad [x]\,E$$

The abstraction rules must by applied in the order they are listed here.

$$\begin{array}{lll}
[x]\,(E_1\,E_2) & \Rightarrow \mathbf{S}\,([x]\,E_1)\,([x]\,E_2) & \\
[\mathbf{P}\,x\,y]\,E & \Rightarrow \mathbf{U}\,([x]\,([y]\,E)) & \\
[x]\,x & \Rightarrow \mathbf{I} & \\
[x]\,y & \Rightarrow \mathbf{K}\,y & \text{, where y is a constant or a variable other than x.}
\end{array}$$

Local definitions must be removed from the expression before the abstraction is done. The algorithm to remove local definitions is explained by showing how two local definitions with one argument each, are removed.

$$\begin{array}{llll}
E_1 & \mathbf{where}\ f\,x & = E_2 & \\
 & \phantom{\mathbf{where}}\ g\,y & = E_3 & \text{the arguments are removed first}
\end{array}$$

$$\begin{array}{lllcll}
\rightarrow & E_1 & \textbf{where}\ f & = & [x]\,E_2 & \\
 & & g & = & [y]\,E_3 & \text{then the definitions are paired} \\
\rightarrow & E_1 & \textbf{where}\ (f{:}g) & = & ([x]\,E_2{:}[y]\,E_3) & \text{next, the recursion is handled} \\
\rightarrow & E_1 & \textbf{where}\ (f{:}g) & = & \mathbf{Y}\,([(f{:}g)]\,([x]\,E_2{:}[y]\,E_3)) & \text{lastly the definition is removed} \\
\rightarrow & \multicolumn{5}{l}{([(f{:}g)]\,E_1)\ (\mathbf{Y}\,([(f{:}g)]\,([x]\,E_2{:}[y]\,E_3)))}
\end{array}$$

The improvement rules to reduce the size of an expression are

$$\begin{array}{lll}
\mathbf{S}\,(\mathbf{K}\,E_1)\,(\mathbf{K}\,E_2) & \Rightarrow \mathbf{K}\,(E_1\,E_2) & \\
\mathbf{S}\,(\mathbf{K}\,E_1)\,\mathbf{I} & \Rightarrow E_1 & \\
\mathbf{S}\,(\mathbf{K}\,E_1)\,E_2 & \Rightarrow \mathbf{B}\,E_1\,E_2 & \text{, if no earlier rule applies} \\
\mathbf{S}\,E_1\quad(\mathbf{K}\,E_2) & \Rightarrow \mathbf{C}\,E_1\,E_2 & \text{, if no earlier rule applies}
\end{array}$$

The reduction order used in the SASL system is not specified in this formalism.

Suitability of the description method

Although this description method is syntactically appealing and very close to the notation of abstraction used in mathematical textbooks, it has a number of disadvantages. A serious drawback is that the various algorithms introduced have no intrinsic names. The absence of these names turns out to be problematic when it is not quite clear from the context which algorithm is meant and the absence of these names is especially annoying when several algorithms are merged. It is necessary to combine transformations to achieve a more efficient algorithm, or to handle more complex cases. Another limitation of the syntactical match is that it is only adequate to describe relatively simple transformations. For instance, in the original description the transformations necessary to remove locally defined functions are not defined by some general applicable transformation rule, but they are specified only by an example. The introduction of additional description tools is necessary to define such a transformation rule. Also the entire transformation from SASL to SK-combinator code is not specified as one algorithm, but as a number of transformations and some textual indication how they must be applied.

4.4 Bracket abstraction described by syntactical matching

The obvious way to handle the problem with the names of the transformations is to define the transformations as functions. This spoils the nice syntactical match a little bit, since there are now two kinds of objects; the syntactical objects to transform and the functions performing the transformations. An attempt to keep these matters separated is the use of Scott brackets, ⟦ and ⟧, to indicate the match on syntactical objects. These symbols are widely used in denotational semantics to indicate that syntactical objects are handled [Gordon 79, Schmidt 86, Fehr 89].

The combinator definitions and the algorithm for bracket abstraction shown below are similar to the definitions in textbooks on the implementation of functional programming languages [Peyton Jones 87]. The transformation shown here is extended to deal with SASL expressions instead of λ-terms. The algorithm is expressed in a kind of functional programming language. This language is usually not properly defined, but so simple that no formal definition is felt to be necessary. However, good text books on denotational semantics give a firm definition. The use of this description method as if it were a functional programming language suggests a certain accuracy in the description which is usually not there, e.g. a last alternative to handle the 'do nothing' case is typically written in half of the situations where it is required.

Combinators used in the abstraction algorithm

The combinators are defined by their reduction rules.

```
S f g x    → f x (g x)
K x y      → x
I x        → x
B f g x    → f (g x)
C f g x    → f x g
U f (P x y) → f x y
Y f        → f (Y f)
```

The expressions handled

The expressions handled are SASL expressions in applicative form. The keywords are printed in bold face. Combinators are printed in uppercase. All other expressions are variables: either SASL variables or transformation variables, they are printed in lower case.

The following conventions are used:

e,e_i,p,q	are arbitrary expressions;
f,f_i	are function identifiers, or function identifiers with some formal arguments;
x,y,a_i	are arguments (variables or pairs);
v	is a variable;
cv	is a constant or a variable;
def,def_i	are definitions;
defs,locals	are sequences of (local) definitions.

By applying the transformations described here a SASL expression can be transformed to a pure combinator expression. C⟦ e ⟧ Compiles the expression e, i.e. transforms it to a combinator term. The compilation algorithm is described by a number of transformation rules. The first transformation rule alternative applicable must be used.

$$\begin{aligned}
\mathrm{C}\llbracket\, e_1\, e_2 \rrbracket &= \mathrm{C}\llbracket\, e_1 \rrbracket\, \mathrm{C}\llbracket\, e_2 \rrbracket \\
\mathrm{C}\llbracket\, e \ \textbf{where}\ \mathrm{locals} \rrbracket &= \textbf{Rls}\ e\ \llbracket\, \mathrm{locals} \rrbracket \\
\mathrm{C}\llbracket\, c \rrbracket &= c
\end{aligned}$$

The abstraction algorithm

To avoid unnecessary scanning of the expression tree the optimization rule is invoked whenever appropriate during the abstraction. In the previous description this could not be indicated since the optimization rule has no name in that formalism.

$\mathbf{A}\ x\ \llbracket\, e \rrbracket$ Abstracts the argument x from the expression e.

$$\begin{aligned}
\mathbf{A}\ x\ \llbracket\, e \ \textbf{where}\ \mathrm{locals} \rrbracket &= \mathbf{A}\ x\ \llbracket\, \textbf{Rls}\ e\ \llbracket\, \mathrm{locals} \rrbracket \rrbracket \\
\mathbf{A}\ x\ \llbracket\, e_1\, e_2 \rrbracket &= \textbf{Opt}\llbracket\, \mathrm{S}\ (\mathbf{A}\ x\ \llbracket\, e_1 \rrbracket)\ (\mathbf{A}\ x\ \llbracket\, e_2 \rrbracket) \rrbracket \\
\mathbf{A}\ (\mathrm{P}\ x\ y)\ \llbracket\, v \rrbracket &= \mathrm{U}\ (\mathbf{A}\ x\ (\mathbf{A}\ y\ \llbracket\, v \rrbracket)) \\
\mathbf{A}\ x\ \llbracket\, x \rrbracket &= \mathrm{I} \\
\mathbf{A}\ x\ \llbracket\, cv \rrbracket &= \mathrm{K}\ cv
\end{aligned}$$

$\mathbf{Aa}\llbracket\, f \rrbracket$ Abstracts all arguments from a function definition.

$$\begin{aligned}
\mathbf{Aa}\llbracket\, \mathrm{P}\ a_1\ a_2 = e \rrbracket &= \mathrm{P}\ a_1\ a_2 = e \\
\mathbf{Aa}\llbracket\, f\ a = e \rrbracket &= \mathbf{Aa}\llbracket\, f = \mathbf{A}\ a\ \llbracket\, e \rrbracket \rrbracket \\
\mathbf{Aa}\llbracket\, \mathrm{def} \rrbracket &= \mathrm{def}
\end{aligned}$$

$\textbf{Rls}\ e\ \llbracket\, \mathrm{locals} \rrbracket$ Removes the local definitions from the expression e. Arguments of the definitions are removed and they are grouped to a single definition. Then this definition is removed.

$$\textbf{Rls}\ e\ \llbracket\, \mathrm{locals} \rrbracket = \textbf{Rl}\ e\ \llbracket\, \textbf{Col}\llbracket\, \textbf{Aal}\llbracket\, \mathrm{locals} \rrbracket \rrbracket \rrbracket$$

$\textbf{Rl}\ e\ \llbracket\, \mathrm{locals} \rrbracket$ Removes a single local definition from the expression e. Recursion in the definition is removed by a Y combinator and abstracting the name of the function from its body.

$$\textbf{RL}\ e_1\ \llbracket\, f = e_2 \rrbracket = (\mathbf{A}\ f\ \llbracket\, e_1 \rrbracket)\ (\mathrm{Y}\ (\mathbf{A}\ f\ \llbracket\, e_2 \rrbracket))$$

$\textbf{Aal}\llbracket\, \mathrm{defs} \rrbracket$ Abstract the arguments from each definition in a list of definitions.

$$\textbf{Aal}\llbracket\, \mathrm{def}_1 \ldots \mathrm{def}_n \rrbracket = \mathbf{Aa}\llbracket\, \mathrm{def}_1 \rrbracket \ldots \mathbf{Aa}\llbracket\, \mathrm{def}_n \rrbracket$$

$\textbf{Col}\ \llbracket\, \mathrm{defs} \rrbracket$ Collects a list of local definitions to a single definition. Definitions are paired from back to front.

```
Col ⟦ f1     = e1
      ...
      fn-1   = en-1
      fn     = en    ⟧ = Col ⟦ f1          = e1
                               ...
                               P fn-1 fn  = P en-1 en ⟧
Col ⟦ def            ⟧ = def
```

Improving the generated combinator code

Opt⟦ e ⟧ Optimizes the expression e. It is invoked by the abstraction rule. So, no scanning of the expression tree is needed.

```
Opt⟦ S (K e1) (K e2) ⟧   = K (e1 e2)
Opt⟦ S (K e1) I ⟧        = e1
Opt⟦ S (K e1) e2 ⟧       = B e1 e2
Opt⟦ S e1 (K e2) ⟧       = C e1 e2
Opt⟦ e ⟧                 = e
```

Reduction

Reduction is specified by the semantics of the combinator expressions. The order in which the rewrite rules are applied to the various parts of the expression is determined by the normal order reduction strategy. This means that the leftmost redex is rewritten. Sometimes it is necessary to reduce some subexpressions before the topmost redex term can be rewritten. For instance, before an addition can be performed both arguments must be reduced. Two functions are used to specify the reduction process. **Red**uce determines the order of reductions. First the left sub-tree of an application node is reduced. Then **Rew**rite is applied to the top node. This function tries to apply the reduction rules for the combinators, after a rewriting the reduction is continued when appropriate. These functions will transform an expression until the top cannot be rewritten any more. Then the expression is said to be in head normal form, **hnf**.

```
Red⟦ e1 e2 ⟧             = Rew⟦ Red⟦ e1 ⟧ e2 ⟧
Red⟦ e ⟧                 = e

Rew⟦ S e1 e2 e3 ⟧        = Red⟦ e1 e3 (e2 e3) ⟧
Rew⟦ K e1 e2 ⟧           = Red⟦ e1 ⟧
Rew⟦ I e ⟧               = Red⟦ e ⟧
Rew⟦ B e1 e2 e3 ⟧        = Red⟦ e1 (e2 e3) ⟧
Rew⟦ C e1 e2 e3 ⟧        = Red⟦ e1 e3 e2 ⟧
Rew⟦ U e1 e2 ⟧           = U e1 Red⟦ e2 ⟧
Rew⟦ EQ e1 e2 ⟧          = Red⟦ e1 ⟧ = Red⟦ e2 ⟧
Rew⟦ PLUS e1 e2 ⟧        = Red⟦ e1 ⟧ + Red⟦ e2 ⟧
```

$$\begin{aligned}
\mathbf{Rew}[\![\, \text{MINUS } e_1\, e_2 \,]\!] &= \mathbf{Red}[\![\, e_1 \,]\!] - \mathbf{Red}[\![\, e_2 \,]\!] \\
\mathbf{Rew}[\![\, \text{TIMES } e_1\, e_2 \,]\!] &= \mathbf{Red}[\![\, e_1 \,]\!] \times \mathbf{Red}[\![\, e_2 \,]\!] \\
\mathbf{Rew}[\![\, \text{COND } e_1\, e_2\, e_3 \,]\!] &= \mathbf{IF}\ \mathbf{Red}[\![\, e_1 \,]\!]\ e_2\, e_3 \\
\mathbf{Rew}[\![\, e \,]\!] &= e \\
\\
\mathbf{U}\ e_1\ [\![\, \text{P } e_2\, e_3 \,]\!] &= \mathbf{Red}[\![\, e_1\, e_2\, e_3 \,]\!] \\
\\
\mathbf{IF}\ \text{TRUE } e_1\, e_2 &= \mathbf{Red}[\![\, e_1 \,]\!] \\
\mathbf{IF}\ \text{FALSE } e_1\, e_2 &= \mathbf{Red}[\![\, e_2 \,]\!]
\end{aligned}$$

The strategy and the reduction rules are combined into a single algorithm to obtain a feasible implementation. This is similar to the combination of the functional reduction strategy and the rewrite rules for Clean described in the next chapter. It is possible to specify these algorithms separately, but that involves a lot of overhead to scan the tree for the next redex and some additional constructors to pass information from the rewrite algorithm to the strategy. This would resemble very much to the two level approach proposed by van Eekelen and Plasmeijer [van Eekelen 86, 88].

Suitability of the description method

Unfortunately there is no strict separation between syntactical objects and the transformation functions in the description above. In the rules for abstraction, algorithmical components are used inside the syntactical region without being syntax. The expressions must be evaluated to become syntactical objects. Sometimes the symbol ≡ is used instead of = to indicate that the right hand side of the rule can be used as a syntactical object. In the **A**bstract arguments rule, the arguments, a_i, are taken from the syntactical region to the transformation region without notice. Also matching arbitrary variables and specific variables is somewhat tricky. The difference between e_1, e_2 and x and v in the abstraction rule is not very clear. The fact that v is a variable in the rule to abstract a pair is quite subtle. In the abstraction rule there is even a match on syntactical structure outside the brackets. Sometimes uppercase characters are used for constants and lower case characters for variables, but this cannot be used consistently. The syntactical match can be used for a single context free transformations of languages with a relative simple structure, but for richer languages and more complex transformations it becomes very tricky. For instance, in the translation of Miranda to Clean it is necessary to distinguish between an arbitrary argument, an argument that is a variable, an argument that is a variable equal to an other argument, an argument that is an arbitrary constructor with a variable number of sub-arguments, an argument that is a member of a specific class of predefined values and an argument that is a specifc symbol [Koopman 88].

4.5 Bracket abstraction as abstract program transformation

Since it appears to be impossible to keep the objects handled and the algorithm separated we propose to express both in a functional programming language. The essential differences between the description employing syntactical matching and the description in a functional language is, that the language constructs to be transformed are represented as syntactical objects between ⟦ and ⟧ in the description above, while they are represented by a data structure in the functional description.

The expressions handled

The SASL expressions are represented by an AST. The Miranda data structure expr is used to hold it.

```
expr                                  || An expression is:
 ::=  AP expr expr                 |  || an application of one expression to another one;
      WHERE expr [def]             |  || an expression with local function definitions;
      VAR var                      |  || a variable; or one of the following constants:
      INT num                      |  || the integers;
      BOOL bool                    |  || the booleans;
      S |K   |I   |B   |C          |  || the combinators to distribute arguments;
      Y                            |  || the combinator to handle recursion;
      P |U   |NIL                  |  || the constructors for lists;
      COND                         |  || the conditional;
      EQ | PLUS | MINUS | TIMES       || some other curried operators.
```

A definition has a left-hand side, lhs, that is either a variable with a sequence of arguments or a pair. An argument is either a variable or a pair of arguments. In order to avoid the construction of an additional data-type it is represented by an expression. The right hand side, rhs, of a definition may be any expression.

```
def  ::=  DEF lhs rhs

lhs  ==   expr                        || A lhs is a pair or a var with arguments.
rhs  ==   expr                        || All expressions are allowed on the rhs
var  ==   [char]
```

Combinators used in the abstraction algorithm

The rewrite behaviour of the combinators defined is specified by the function rewrite. The combinators themselves can not be functions since trees which are not in head normal form must be handled (e.g. by the optimization rules). So, an interpreting function is used.

```
rewrite :: expr -> expr
rewrite (AP (AP (AP S f) g) x)      = AP (AP f x) (AP g x)
rewrite (AP (AP K x) y)             = x
rewrite (AP I x)                    = x
rewrite (AP Y f)                    = AP f (AP Y f)
rewrite (AP (AP (AP B f) g) x)      = AP f (AP g x)
rewrite (AP (AP (AP C f) g) x)      = AP (AP f x) g
rewrite (AP (AP U f) expr)          = unpair f (reduce expr)
rewrite expr                        = expr

unpair f (AP (AP P x) y))           = AP (AP f x) y
```

The rule for U is special since it requires the reduction of a sub-term before it can be applied. Basic operations like PLUS, MINUS, TIMES, EQ and COND, also require reduction before they can be applied. The function reduce evaluates an expression tree to head normal form. It is defined below.

The abstraction algorithm

By applying the transformation rules, an expression representing SASL program is transformed to a pure combinator expression. This process is initiated by the function compile.

```
compile :: expr -> expr
compile (AP e1 e2)                  = AP (compile e1) (compile e2)
compile (WHERE e defs)              = optimize (remove_where e defs)
compile expr                        = expr
```

The abstraction algorithm expressed in Miranda reads:

```
abstract :: expr -> expr -> expr
abstract x (WHERE e defs)           = abstract x (remove_where e defs)
abstract x (AP e1 e2)               = AP( AP S (abstract x e1)) (abstract x e2)
abstract (AP (AP P x) y)(VAR z)     = AP U (abstract x (abstract y (VAR z)))
abstract var exp                    = I            , if var = exp
                                    = AP K exp     , otherwise
```

Locally defined functions are removed from an expression as outlined above.

```
remove_where :: expr -> [def] -> expr
remove_where expr defs
 = AP (abstract f expr) (AP Y (abstract f body))
   where   DEF f body = collect (map abs_args defs)
```

Arguments are removed from a function definition by abstracting them from the function body. As a special case the left hand side of a definition can also be a pair.

```
abs_args :: def -> def
abs_args (DEF (AP (AP P h) t) body)  = DEF (AP (AP P h) t) body
abs_args (DEF (AP f x)        body)  = abs_args (DEF f (abstract x body))
abs_args (DEF f               body)  = DEF f body
```

The sequence of local function definitions is combined to one definition by pairing them from back to front.

```
collect :: [def] -> def
collect (def:defs)
 = def              , if defs = [ ]
 = DEF f body       , otherwise
   where  f              = AP (AP P f1) f2
          body           = AP (AP P body1) body2
          DEF f1 body1 = def
          DEF f2 body2 = collect defs
```

The resulting AST fits nicely in the data structure expr, but variables and local definitions do not occur.

Improving the generated combinator code

The function optimize applies the improve rules to every application, starting in the leaves of the expression tree. The optimization rules are not included in the abstraction algorithm to enable the observation of the effect of these improvements.

```
optimize :: expr -> expr
optimize (AP e1 e2)   = improve (AP (optimize e1) (optimize e2))
optimize expr         = expr

improve :: expr -> expr
improve (AP (AP S (AP K e1)) (AP K e2))  = AP K (improve (AP e1 e2))
improve (AP (AP S (AP K e1)) I)          = e1
improve (AP (AP S (AP K e1)) e2)         = AP (AP B e1) e2
improve (AP (AP S e1) (AP K e2))         = AP (AP C e1) e2
improve expr                             = expr
```

Executing the generated combinator code

By applying the given rewrite rules the value of the resulting combinator expression can be computed. Two functions are used to describe the reduction to hnf. Reduce controls the global order of reductions and rewrite performs the reductions, after a rewrite the reduction is continued when appropriate. Rewrite invokes the required reductions for rewrite rules that require arguments in hnf, a function corresponding to such a top node handles the reduction according to that node.

```
reduce :: expr -> expr
reduce (AP e1 e2)                 = rewrite (AP (reduce e1) e2)
reduce exp                        = exp

rewrite :: expr -> expr
rewrite (AP (AP (AP S f) g) x)    = reduce (AP (AP f x) (AP g x))
rewrite (AP (AP K x) y)           = reduce x
rewrite (AP I x)                  = reduce x
rewrite (AP Y f)                  = reduce (AP f (AP Y f))
rewrite (AP (AP (AP B f) g) x)    = reduce (AP f (AP g x))
rewrite (AP (AP (AP C f) g) x)    = reduce (AP (AP f x) g)
rewrite (AP (AP U f) exp)         = unpair f (reduce exp))
rewrite (AP (AP PLUS x) y)        = plus  (reduce x) (reduce y)
rewrite (AP (AP MINUS x) y)       = minus (reduce x) (reduce y)
rewrite (AP (AP TIMES x) y)       = times (reduce x) (reduce y)
rewrite (AP (AP EQ x) y)          = eq (reduce x) (reduce y)
rewrite (AP (AP (AP COND c) t) e  = cond (reduce c) t e
rewrite exp                       = exp

unpair  f         (AP (AP P x) y) = reduce (AP (AP f x) y)
plus    (INT i1)  (INT i2)        = INT  (i1 + i2)
minus   (INT i1)  (INT i2)        = INT  (i1 - i2)
times   (INT i1)  (INT i2)        = INT  (i1 * i2)
eq      e1        e2              = BOOL (e1 = e2)
cond    (BOOL True )    t  e      = reduce t
cond    (BOOL False)    t  e      = reduce e
```

The reduction rules defined here perform a correct string reduction of the expression. A more efficient implementation of the reduction of these expressions will use graph reduction. Using graph reduction every sub-expression is reduced at most once.

Suitability of the description method

The overhead involved in implementing the abstraction algorithm in a functional language appears to be very limited. This implementation effort is rewarded by a very clear and executable specification that can be partially checked by a Miranda implementation. The tree-like data structure needed to represent the SASL expressions appears to be essential in the abstraction process.

4.6 Discussion

In this chapter three descriptions of the language transformation known as bracket abstraction are compared. The two conventional descriptions use special purpose notations for the descriptions and are hence syntactically appealing. A drawback of the first description is that it lacks intrinsic names for the individual transformations. Only a set of transformation rules is given, but the combination of these rules to a complete algorithm cannot be described in this formalism. The second description possesses names for the individual transformations, the language to transformed is distinguished from the description language by surrounding it with special symbols. Unfortunately, it is generally not possible to make such a separation between both languages using this notation.

Using a programming language for the specification has the advantages, as said before, that the description can be partially checked by the implementation of the description language and that it is executable. This is a valuable test for the combinator code generated; the code generated is quite unreadable for human beings. Also the effect of the improvement rules can be determined easily using an executable description; apply the function optimize to some (generated) expressions. A possible drawback of such a description could be the implementation overhead involved, since no special purpose syntax can be introduced the description might become very long. But, we have shown that this overhead is very small when the functional programming language Miranda is used as description formalism. The obtained description is at least as clear as the mathematical descriptions.

In conclusion it can be stated that functional descriptions are very convenient to express these kind of program transformations. Descriptions given in this way tend to be correct, complete, compact, clear and executable. In the next chapter an abstract program transformation will be given for a more complex translation.

Chapter 5
Translating Clean to ABC-Code

In this chapter a translation of Clean to ABC-code is treated as a real life example of the abstract program transformation method introduced in the previous chapter. This transformation is rather complex since the relatively high-level declarative language Clean is transformed into a low-level imperative assembly language.

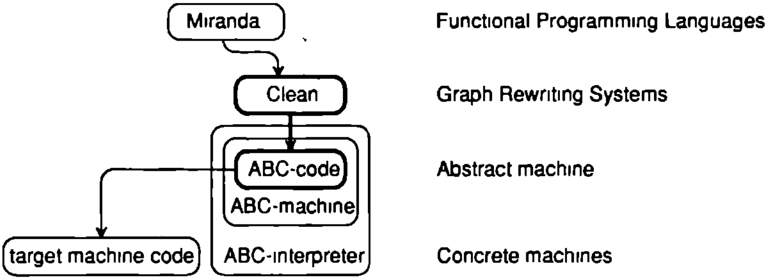


Fig 5.1 Transformation of Clean to ABC-code as a phase in the compilation process.

The goal of the translation is, of course, to transform Clean programs to equivalent and efficient ABC-programs. Since ABC-code is translated to (or interpreted by) various concrete machines it is not possible to assign execution times to the instructions, hence it is impossible to develop an optimal compilation scheme.

To increase the performance of the generated code some deviations of the operational semantics of Clean [Eekelen 89] are made. Although Clean graphs are mapped directly to graphs in the graph store of the ABC-machine this store is not updated after every rewrite step. During the rewriting the information is stored on the stacks and (implicit) in the instruction sequence executed. The performance of ABC-programs heavily manipulating the graph store for simple computations will be low compared to ABC-code that employs the B-stack whenever appropriate. One of the ways to use the B-stack is outlined.

Section 5.1 contains an informal description of graph rewriting in Clean. In section 5.2 graph rewriting on the ABC-machine and the calling conventions are described informally. In section 5.3 the AST to hold the rewrite rules is defined. The code produced by the translation scheme is ABC-assembler which was introduced in chapter 3. The basic translation scheme is described in 5.4. In section 5.5 the runtime system is discussed. The way curried functions are implemented is outlined in section 5.6. Finally, a number of optimizations is discussed. The most important optimization is the use the B-stack to pass strict arguments of a basic type to functions.

5.1 Graph rewriting in Clean

To understand the code generation schemes discussed in this chapter it is important to have a proper view of the operational behaviour of a Clean program. A complete description of Clean is outside the scope of this work, we refer to [Barendregt 87b, Brus 87, Eekelen 89]. The semantics of Clean will be explained informally below.

Clean is a lazy higher order functional programming language based on Functional Graph Rewriting Systems, it is designed as intermediate language between arbitrary functional languages and (sequential) machines. A **Functional Graph Rewriting System** (*FGRS*) is a Graph Rewriting System using the functional reduction strategy [Eekelen 88]. A reduction strategy is a function indicating the *redex* to be rewritten. The **functional strategy** delays the reduction of an expression until its value is needed. However, an expression is always reduced before its value is used. The functional strategy in Clean prescribes the same evaluation order as used in Miranda: **lazy evaluation**. A **Graph Rewriting System** (*GRS*) is an extension of Term rewriting Systems where the terms are replaced by directed graphs in order to avoid the duplication of work via sharing of expressions [Barendregt 87a, 87b, 88]. A **Term Rewriting System** (*TRS*) is a computational paradigm consisting of a collection of rewrite rules to transform terms (expressions) into equivalent terms [Klop 87].

A Clean program consists of a set of (typed) graph rewrite rules. The type system is based on the Milner/Mycroft type inference scheme [Milner 78, Mycroft 84].

A subgraph is a **redex** (*red*ucable *ex*pression) if there is a **left hand side** (*lhs* or *pattern*) of rewrite rule that matches this graph. A match is a mapping from the pattern to the graph that is the identity on constants and preserves the node structure. A graph is in **root normal form** (*rnf*) if the whole graph is not a redex and will never become a redex. A graph is in **normal form** (*nf*) if it does not contain any redex.

The rewrite rules are used to reduce the initial graph containing the symbol Start to normal form. The functional reduction strategy is used: rewrite alternatives are tried in textual order; patterns are matched from left to right; evaluation to root normal form is forced before an actual argument is compared with non-variable part of the pattern. An arguments is **strict** when its value always is needed in the reduction of the function. It is possible to deviate from the functional strategy by adding strictness annotations, denoted by I, to the rewrite rules. Such a graph is reduced eagerly to root normal form.

A redex is rewritten by constructing the graph specified in the **right hand side** (*rhs*) of the rule: the **contractum**. Then all references to the root redex are *redirected* to the root of the contractum. There are also rewrite rule alternatives consisting of a redirection only; no contractum is specified in these rules. Nodes that cannot be reached from the root of the graph are **garbage**, they must be removed from the graph.

A small example is used to illustrate graph reduction in Clean. The example is even so small that no sharing of computations occurs. The Start rule initiates the computation of the length of a list. The rule Length takes two arguments; a number to record the number of list elements scanned and the list to be scanned.

```
:: Start -> INT                                ;   || The type of the start rule
   Start -> Length 0 (Cons 3 (Cons 4 Nil))     ;   || The rewrite rule for Start

:: Length ! INT I (List x)   -> INT            ;   || Both arguments are strict
   Length n (Cons a b)       -> Length (+I n 1) b  |   || +I is a delta rule to add
   Length n Nil              -> n              ;   || 2nd alternative; a redirection
```

The reduction process is illustrated by the following rewriting sequence (the redex rewritten is underlined):

```
  Start                                 || a: This is the only redex, apply Start rule
→   Length 0 (Cons 3 (Cons 4 Nil))      || b: This graph as a whole is the new redex
→   Length ( +I 0 1 ) (Cons 4 Nil)      || c: The strict arguments are reduced first
→   Length 1 (Cons 4 Nil)               || d: Rewrite according to first alternative
→   Length ( +I 1 1 ) Nil               || e: Again one strict argument to be reduced
→   Length 2 Nil                        || f: 1st alternative does not match; use 2nd
→   2                                   || g: This graph is in normal form
```

This rewriting process is depicted below. The graphs correspond to each of the steps shown above. The garbage that results from one rewrite step is drawn gray, it is removed in the next snapshot. The dataroot is usually not shown, but it is included here to show the redirections clearly.

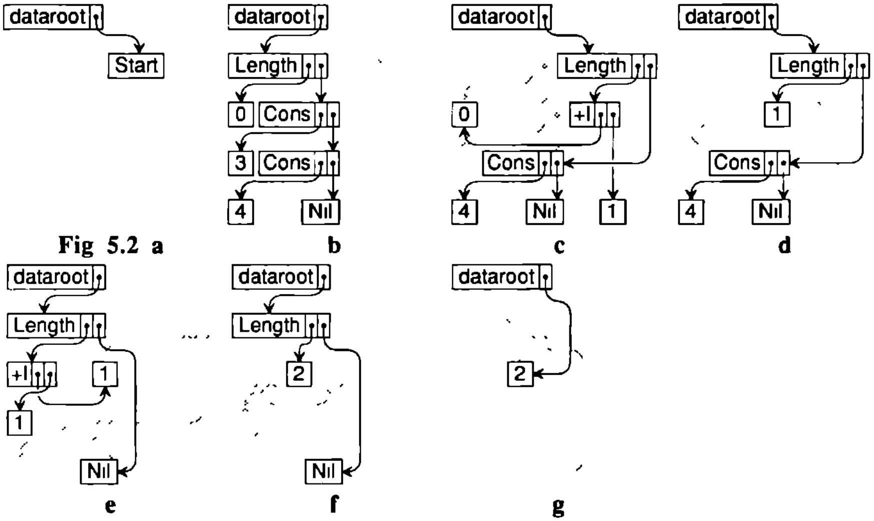


Fig 5.2

It is important to mention that Clean is a **higher order** language: symbols can be used **Curried**; without (some of) their arguments. The delta rule AP supplies one argument to a curried symbol, when all arguments of such a function are gathered its reduction is initiated. This is illustrated by the next example:

```
Twice f x   -> AP f (AP f x)   ;   || Twice applies the function f two times to the argument x
Start       -> Twice ++I 0     ;   || ++I is the delta rule to increment an integer
```

The reduction process is illustrated by the next rewrite sequence:

```
Start                          || The initial graph
→    Twice ++I 0               || The graph after rewriting according to the start rule
→    AP ++I (AP ++I 0)         || This graph will be rewritten by the delta rule AP
→    ++I ( AP ++I 0 )          || AP has supplied the delta rule ++I its argument
→    ++I ( ++I 0 )             || AP has supplied again the argument needed
→    ++I 1                     || The delta rule ++I must be applied again
→    2                         || The graph is reduce to its normal form
```

5.2 Graph rewriting on the ABC-machine

In this section an informal description of graph rewriting on the ABC-machine is given. The functional strategy and the rewrite algorithm are combined into a single piece of ABC-code for every rewrite rule. Associated with each rewrite rule there is a sequence of ABC-instructions which reduce a node in the graph to its root normal form. Using the functional strategy a node must be reduced to rnf, as soon as rewriting is initiated.

A graph reduction program in the ABC-machine consists of:

- rule dependent code to reduce a redex to its rnf according to the functional strategy and the rule in the system;
- code for the predefined rules;
- a **run-time system**: a fixed piece of code that initiates the reduction of the Start rule to normal form and prints the reduct.

For each Clean rule there is ABC-code to:

- prepare the arguments, i.e. construct a stack frame as expected by the code for the rule alternatives;
- match each rule alternative to the actual redex and rewrite accordingly;
- handle the situation that none of the rule alternatives is applicable. Code according to an additional rule alternative is generated for this purpose.

These pieces of code and their **calling conventions** (the interface between caller and callee) will be described below. Arguments and results can be passed among functions in the graph, on the A-stack and on the B-stack. It is more efficient to handle references to objects on the A-stack instead of in nodes of the graph: the references are directly available on the stack. When appropriate it is even more efficient to pass objects on the B-stack: the objects themselves are handled instead of references to the objects. The B-stack is not used in the basic compilation scheme, the description of its use is delayed until section 5.7.

To show the effect of the described actions a running example will be used. The reduction of a node containing the symbol F and two matching arguments will be shown. The rewrite rules used are:

```
::  F ! (List x) ! (List x)            -> INT                 ;
    F (Cons a (Cons b c)) (Cons d e)   -> G f f , f: Cons b e ;

::  G (List x) (List x)  -> INT   ;
    G a b                -> 1     ;
```

The overall assumption is that once initiated the reduction of a graph continues until it is in rnf. Each rule alternative expects an A-stack frame containing a reference to the redex currently reduced and the node-id's of all arguments. The reference to the first argument is on top of the stack.

Construction of the stack-frame

The reduction according to the rewrite rule can be initiated in two different ways. The preparation of the arguments needed depends on the calling circumstances.

The first situation is the initiation of the reduction by a jsr_eval instruction. In this situation the stack-frame consists only of a reference to the redex (see the definition of this instruction in chapter 3). To construct the desired stack frame the node is marked as being under reduction, the arguments are pushed on the stack and the strict ones are reduced to rnf. The start of this code segment is indicated by the **node entry**.

In the second situation the reduction is initiated by a curried application of the function. Here, the node-id of the redex and the arguments will be on the stack, they are gathered by the code for the AP-rule. Strict arguments must still be reduced, so the code for the **apply entry** hooks up there in the code to prepare arguments.

In the figures below the relevant parts of the graph store and the A-stack frame are shown for a rewrite according to the rewrite rule above. New parts in the pictures are printed bold. Graphs with unknown contents are drawn as grey boxes.

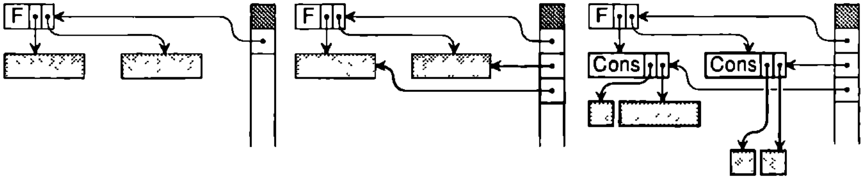


Fig 5.3.a **b** **c**

a: The A-stack frame and graph upon entrance of the node entry.

b: The stack frame is extended with the node-id's of the arguments at the the apply entry.

c: The state expected by the rule alternatives; both arguments are strict and hence reduced to rnf.

Rule alternative entries

For each alternative there is a separate **rule alternative entry**. The code for each entry consists of a matching phase which determines whether this alternative is applicable and a rewriting phase which performs the rewrite. When an alternative appears to be inapplicable, execution proceeds with the next one. Due to the generation of an additional alternative there is always a next one.

Matching

Arguments are matched from left to right. If the formal argument is a variable the argument trivially matches. Otherwise, the argument is brought in root normal form and the symbol in the graph is compared with the specified symbol. When the symbol appears to match, its sub-arguments are pushed on the A-stack and matched themselves. If one of the arguments does not match the pattern, all sub-arguments are popped off the stack and execution proceeds with matching the next rule alternative. When the matching of all arguments succeeds the rewrite must be performed.

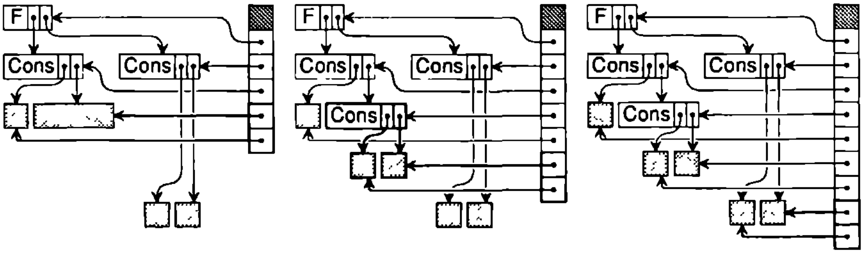


Fig 5.3.d e f

d: The situation after matching the top level of the first argument and pushing its sub-arguments.

e: The second sub-argument of the first argument is reduced and matched. After the successful match the node-id's of the sub-arguments are pushed on the stack.

f: The second argument also matches. So, this rule alternative is applicable.

Rewriting

To increase the performance of the generated code some deviations of the operational semantics of Clean [Barendregt 87] are made. According to the semantics of Clean the contractum must be constructed (if present in the rule) and all references to the redex must be redirected to the contractum. Redirections are conceptually elegant and explain the semantics of graph rewriting very well. A straightforward, but also inefficient, implementation of redirection would have to examine the whole graph; every reference to the root of a redex is substituted by a reference to the contractum. A more efficient implementation is achieved by overwriting the redex with the root of the contractum. Although Clean graphs are mapped directly to graphs in the graph store of the ABC-machine this store is not updated after every rewrite step, but only when a root normal form is reached. During the rewriting the information is stored on the stacks and (implicit) in the instruction sequence executed. The redirection is implemented efficiently by overwriting the root node of the redex with the root of the contractum. In this way all references to the redex are automatically redirected to the contractum. Three situations are distinguished; the contractum is the next redex, the specified contractum is a root normal form and no contractum is specified in this rewrite alternative.

Redirection to a reducible contractum

When the contractum is not known to be in rnf, the rewrite rule associated with the symbol in the root must be applied. It makes no sense to fill the redex with the root of the contractum. The new rewrite rule will unpack this node immediately since the functional strategy will indicate this redex as the one to rewrite. So, it is possible to delay the update of the graph store until a rnf is reached. The code for this kind of rewrites constructs the stack frame for the first alternative of the new rewrite rule and proceeds immediately with that alternative.

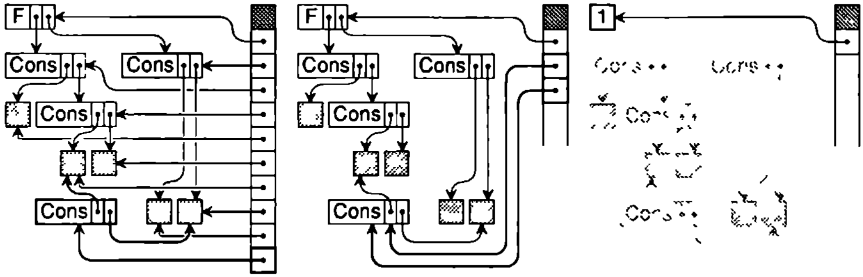


Fig 5.3.g h i

g: The shared node, f, is constructed in the graph store.
h: The stack frame is ready for matching the first rule alternative of G.
i: The reduction according to rule G has been performed; a root normal form is reached.

Redirection to a contractum in root normal form

When the contractum is a rnf, the root node of the redex is overwritten with the root of the contractum. The node containing the redex is indicated by the bottom of the A-stack frame. Rewriting to root normal form is completed. See figure 5.3.i for an example.

By overwriting the redex with its rnf all references to the redex are automatically redirected to the contractum; a very cheap and commonly used implementation of redirection. On the ABC-machine it is always possible to overwrite a node with arbitrary new contents. For an other machine without variable sized nodes the redex is usually overwritten by an **indirection node**; a node indicating where this node actually can be found. A serious drawback of indirection nodes is the testing on these indirections required, even if a node is known to be in rnf.

Redirection to an existing graph

An existing graph cannot be rerooted at some other node without searching the whole graph or using indirection nodes. Since we want to avoid both, a copy of the root node is made. The redex and the top node of the graph are potentially both shared, so copying the top node to the redex immediately might result in the duplication work. To avoid the duplication of work the graph is reduced to rnf before the root node is copied. The graphs remain equal modulo unravelling which is sufficient for term graph rewriting and so sufficient for Clean. The obtained graph is different from the one obtained according to ordinary graph rewrite semantics.

To illustrate these graph manipulations an example is shown.

```
:  Tl ! (List x)     -> (List x)  ;
   Tl (Cons h t)     -> t         ;  || a redirection

:  F -> List INT      ;
   F -> Cons 1 F      ;
```

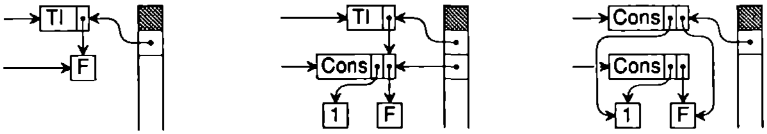


Fig 5.4.a b c

a: Starting the reduction of the graph Tl F. State at the node entry. The arrows from outside indicate sharing.
b: After the reduction of the argument; state at the rewrite entry.
c: After copying the top node. Both nodes contain the proper rnf.

Example of the rewriting process

As example a rewrite rule is considered that computes the length of a list. The equivalent ABC-program generated by the specification presented in section 5.4 is listed (comments are supplied by hand). This is an important example since it illustrates the code generated by the basic compilation scheme.

```
:  Length ! INT ! (List x)   -> INT                 ;
   Length n (Cons a b)       -> Length (+I n 1) b   |
   Length n Nil              -> n                   ;
```

The corresponding ABC-code:

```
[             Descriptor
"Length"      "a_Length" 2 "Length"          ,  || The generated descriptor
              Label                             || The node entry: preparing the arguments
"n_Length",Set_entry        "_cycle" 0       ,  || Mark node to detect cyclic computations
              Push_args     0 2 2            ,  || Push the arguments
              Label                             || The apply entry: reduce strict arguments
"a_Length", Jsr_eval                         ,  || Reduce first argument to rnf
              Push_a        1                ,  || Copy second argument to top of stack
              Jsr_eval                       ,  || Reduce it to rnf
              Pop_a         1                ,  || Pop duplicated second argument
              Label                             || Entry for first rule alternative
"Length1",  Eq_desc_arity   "Cons" 2 1       ,  || Match second argument
              Jmp_false     "Length2"        ,  || Continue with next alternative if match fails
              Push_args     1 2 2            ,  || Push sub-arguments;
              Push_a        1                ,  || Rewrite according to alternative 1; Push b
              Jsr_eval                       ,  || Reduce it
              Create                         ,  || Node for result of +I n 1
```

```
             Create                             , || Node for 1
             FillI            1 0               , || Fill this node
             Push_a           5                 , || Push n. It is known to be a rnf
             Jsr              "+I1"             , || Reduction of +I n 1
             Update_a         1 5               , || Adapt stack frame for call of Length
             Update_a         0 4               , ||
             Pop_a            4                 , || Remove old (sub-)arguments
             Jmp              "Length1"         , || Continue with alternative 1 of Length
             Label                                || Entry for second rule alternative
 "Length2",  Eq_desc_arity    "Nil" 0 1         , || Match argument
             Jmp_false        "Length3"         , || Continue with next alternative if match fails
             Fill_a           0 2               , || Rewrite according to alternative 2;copy node
             Pop_a            2                 , || Remove arguments from stack
             Rtn                                , || Rnf reached
             Label                                || Generated entry for additional alternative
 "Length3",  Jmp              "type_error"      ] || This must be a type error!
```

In the figures below several snapshots of the ABC-machine state are shown during the reduction of a graph according to the code for the rewrite rule Length.

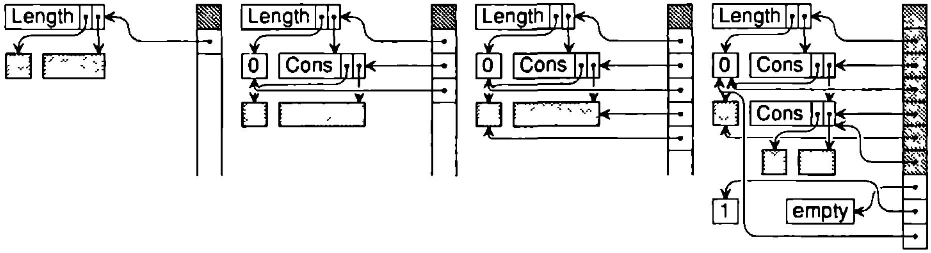
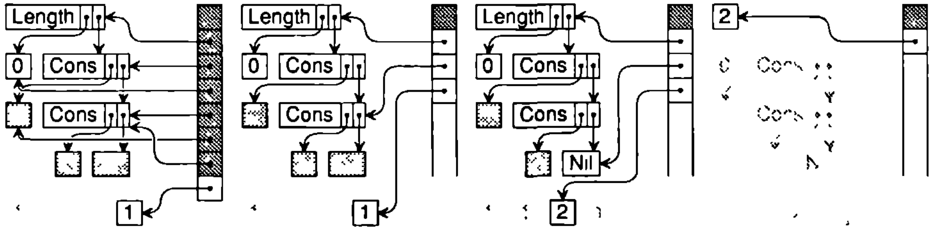


Fig 5.5.a **b** **c** **d**

a: Initial state at node entry.
b: Initial state at first rewrite alternative entry.
c: After successful match of first rule alternative.
d: Reduction according to the first rule alternative. First, the last strict argument is reduced. The state before calling +I1 to reduce first strict argument is shown.



e **f** **g** **h**

e: State returned by +I1. Garbage nodes are grey.
f: State before calling Length1 recursively.
g: State before last call of Length1.
h: State of ABC-machine after reaching the rnf.

5.3 The AST for Clean

The Clean program to be transformed is represented by an AST as proposed in the previous chapter. The construction (lexical analysis and parsing) of the syntax tree is not treated here, this is done by standard compiler construction techniques [Aho 86]. The AST is assumed to be annotated with type and strictness annotations for the code generation. In Clean these annotations may be present in the program, but they can also be derived automatically by a strictness analyser [Nöcker 90] and by type inferencing [Milner 87, Mycroft 84]. When a symbol is used in a strict context (it is annotated as strict), all its strict arguments are annotated as Strict. The generation of these annotations is not treated here.

To facilitate the description the rewrite rules are assumed to be converted to a standard format. All shared nodes in a right hand side (rhs) are defined by explicit definitions on the textual outermost level of that rhs. They are ordered such that a node is defined in textual order before it is used as an argument in another node definition. This is always possible unless there is a cycle in these node definitions. Such nodes on a cycle are annotated as OnACycle.

To reduce the size of the description only a limited number of basic types is treated. In this chapter only the type INT is covered, in appendix B also booleans are handled. All other basic types can be treated in an analogue way.

```
clean        ==  [rewrite_rule]

rewrite_rule ::= TypedRule   typerule   rule    |  || An ordinary typed rewrite rule.
                 ConsRule    typerule   rule    |  || Rewrite rule for a constructor.
                 UntypedRule            rule    |  || An untyped rewrite rule.
                 TypeRule  typerule                || The definition of a type.

typerule     ==  rule
rule         ==  [rulealt]

rulealt      ::= Rewrite   graph graph          |  || A graph substitution,
                 Redirect  graph annots node_id    || or a redirection.

graph        ==  [node]                            || A graph is a list of nodes

node         ::= Node annots node_id symbol args

annots       ==  [annot]
args         ==  [arg]

annot        ::= Strict                         |  || Strict  argument or node
                 OnACycle                       |  || This node definition is on a cycle
                 IsINT                          |  || Argument or node of type INT
                 IsBOOL                            || Argument or node of type BOOL
```

```
arg       ::=  NodeId       annots node_id    | || An argument is either a node-id,
               Term         annots node         || or a node.

symbol    ::=  Symbol       symbolid          | || Type symbols
               Function     symbolid          | || Symbols with associated rule
               Constructor  symbolid          | || Symbols without associated rule
               INTval       num               | || Values of the basic type INT
               BOOLval      bool                || Values of the basic type BOOL

node_id   ==   [char]
symbolid  ==   [char]
```

A number of access functions with the obvious semantics is defined on this syntax tree. Their definitions can be found in appendix B.

Unfortunately, the Miranda type system does not allow to write just node as second alternative for arg. In the Miranda type scheme the constructor must identify the type of the expression uniquely. So, an additional constructor must be inserted in the AST when an other syntactical clause is included.

5.4 Compilation

In this section a compilation scheme is presented which transforms Clean rewrite rules into ABC-assembler. We assume that the AST represents correct Clean programs. The generated ABC-instruction sequence reflects the sequence of actions required for a rewrite specified above. Since the relevant information is stored in annotations, each rewrite rule can be transformed to ABC-code separately. Section 5.2 contains an example of the code produced by this scheme.

```
compiler :: clean -> abc_assembler
compiler = concat.map compile
```

For each rewrite rule a descriptor and code corresponding to the entries discussed above are generated.

```
compile :: rewrite_rule -> abc_assembler
compile (TypedRule typealts alts)
 = fun_descriptor lhs  ++
   prepare_args lhs    ++
   alt_entries alts 1  ++
   gen_type_error symbol_id (# args) (# alts)
   where  lhs = get_lhs (hd typealts)
          NODE annots nodeid (Function symbol_id) args = hd lhs
compile (ConsRule typealts alts)
 = fun_descriptor lhs  ++
   prepare_args lhs    ++
    alt_entries alts 1    ++
    fill_with_rnf symbol_id (# args) (# alts)
    where   lhs = get_lhs (hd typealts)
            NODE annots nodeid (Function symbol_id) args = hd lhs
compile (UntypedRule alts)
 = fun_descriptor lhs ++
    prepare_args lhs    ++
    alt_entries alts 1    ++
    fill_with_rnf symbol_id (# args) (# alts)
    where   lhs = get_lhs (hd alts)
            NODE annots nodeid (Function symbol_id) args = hd lhs
compile (TypeRule typealts)
 = descriptors typealts
```

Descriptors

Descriptor generation for functions:

```
fun_descriptor :: graph -> abc_assembler
fun_descriptor (NODE annots nodeid (Function symbol_id) args:r)
 = [  Descriptor
          symbol_id
          (apply_label symbol_id)
          (# args)
          symbol_id                                ]
```

For each constructor symbol defined in a type definition, a descriptor is generated. Function symbols defined in a type definition are partial functions, the corresponding descriptor is generated along with the code for that partial function. Constructors can also be used curried, so an apply entry containing fill code is defined for them.

```
descriptors :: rule -> abc_assembler
descriptors (alt:rest)
 = cons_descriptor rhs    ++
   fill_code rhs          ++
   descriptors rest
   where   rhs   =  get_rhs alt
descriptors [ ]
 = []

cons_descriptor :: graph -> abc_assembler
cons_descriptor (NODE annots nodeid (Constructor symbol_id) args:r)
 = [  Descriptor
          symbol_id
          apply_lab
          arity
```

```
            symbol_id                                       ]
      where apply_lab
                  = rnf_lab                                      , if arity = 0
                  = apply_label symbol_id                        , otherwise
            arity = # args
cons_descriptor (NODE annots nodeid (Function symbol_id) args:r)
 = []

fill_code :: graph -> abc_assembler
fill_code (NODE annots nodeid (Constructor symbol_id) args:r)
 = [ Label      (apply_label symbol_id)                     ,
     Fill       symbol_id arity rnf_lab arity               ,
     Rtn                                                    ]  , if arity > 0
 = []                                                          , otherwise
   where arity = # args
fill_code (NODE annots nodeid (Function symbol_id) args:r)
 = []
```

Preparing the arguments

Two entries are generated to prepare arguments. The node entry is the regular entry; the target of the jsr_eval instruction. The apply entry is used by curried function applications.

```
prepare_args :: graph -> abc_assembler
prepare_args (NODE annots nodeid (Function symbol_id) args: alts)
 = [ Label      (node_label symbol_id)                   ,
     Set_entry  cycle_lab 0                              ]  ++
   pushargs 0 arity                                         ++
   [ Label      (apply_label symbol_id)                  ]  ++
   reduce_strict_args args 0
   where  arity = # args
```

pushargs is a function which generates the appropriate instruction when there are arguments to push (arity > 0).

Strict arguments are reduced by generating a Jsr_eval for each argument which is annotated Strict in the type definition. The node entry is the target of the jsr_eval instruction, it assumes that the root node-id is on top of the A-stack, so it must be copied if it is not at the top.

```
reduce_strict_args :: arg's -> a_src -> abc_assembler
reduce_strict_args (arg:args) n
 = reduce_arg n ++   reduce_strict_args args (n+1)         , if is_strict arg
 =                   reduce_strict_args args (n+1)         , otherwise
reduce_strict_args [ ] n
 = []
```

```
reduce_arg :: a_src -> abc_assembler
reduce_arg a_src
 = [  Jsr_eval                                ] , if a_src = 0
 = [  Push_a     a_src                        ,
      Jsr_eval                                ,
      Pop_a    1                              ] , otherwise
```

The code for the first rule alternative is generated next to the code for the apply entry, no instructions are required to achieve the continuation.

Rewrite alternative entries

The sequence of alternatives is decomposed to single alternatives. The alternative number is supplied to generate unique labels.

```
alt_entries :: rule -> num -> abc_assembler
alt_entries (alt:rest) alt_number
 = alt_entry alt alt_number ++  alt_entries rest (alt_number+1)
alt_entries [ ] alt_number
 = []
```

For each rule alternative, matching code and rewrite code will be generated. The code generated for the rhs assumes that the matching succeeded. If the graph does not match this lhs, execution proceeds with the next alternative and the rewrite code is not executed.

```
alt_entry :: rulealt -> num -> abc_assembler
alt_entry (Rewrite lhs rhs) alt_number
 = match_code ++ contractum rhs bindings states asp
   where  (match_code,bindings,states,asp)   =  match lhs alt_number
alt_entry (Redirect lhs annots nodeid) alt_number
 = match_code ++ redirection annots nodeid bindings states asp
   where  (match_code,bindings,states,asp)   =  match lhs alt_number
```

Matching

Many of the subsequent transformation rules not only produce code, but also information of the contents of the stack frame. A compile-time number, called asp (for *A*-stack *p*ointer), is used to record the size of the stack frame. The mapping from node-id's to offsets in the stack frame is recorded in the function bindings. The state of the referenced node is recorded in order to generate more efficient code.

```
bindings  ==  node_id -> offset   || Associates an index in the A-stack frame to a node-id
offset    ==  num                 || An index in the A-stack frame
asp       ==  num
```

```
state    ==  node_id -> status  || Associates the status of the node to a node-id
status                          || The states possible
         ::= Undefined        | || This node-id is not defined in the current scope
             Created          | || The node is created but is still empty
             Unknown          | || The node is filled; it is unknown if the graph is a rnf
             InRnf              || This node is in root normal form
```

```
match :: graph -> num -> (abc_assembler,bindings,states,asp)
match [NODE annots nodeid (Function symbol_id) args] alt_number
 = (code, bindings', states', asp')
   where code      = [ Label     label     ] ++    match_code
         bindings  = bind_args args asp empty_bindings
         states    = record_args args initial_status
         asp       = arity
         arity     = # args
         label     = alt_label symbol_id alt_number
         next_alt  = alt_label symbol_id (alt_number+1)
         (match_code,bindings',states',asp')
                   = match_args args bindings states asp next_alt asp arity
```

The function match_args decomposes the matching into matching arguments one by one. A separate offset of the elements in the stack frame is passed since not every argument has a node-id, hence bindings cannot be used. For each specified pattern the evaluation of the actual argument is forced if necessary, match_arg takes care of the actual matching of the sub-graph.

```
match_args :: arg's -> bindings -> states -> asp -> label -> offset -> arity ->
                                                   (abc_assembler,bindings,states,asp)
match_args (NodeId an nodeid:rest) bindings states asp next offset arity
 = match_args rest bindings states asp next (offset-1) arity
match_args (Term ans (NODE n_ans id sym args):rest) binds states asp next offset arity
 = (code, bindings", states", asp")
   where code   = bring_in_rnf annots a_src   ++   match_code  ++   match_rest
         a_src  = asp - offset
         (match_rest,bindings",states",asp")
                = match_args rest bindings' states' asp' next (offset-1) arity
         (match_code,bindings',states',asp')
                = match_arg (NODE (ans ++ n_ans) id sym args)
                                      binds (record id InRnf states) asp next a_src arity
match_args [ ] bindings states asp next_alt offset arity
 = ([ ],bindings,states,asp)
```

The transformation rule match_arg generates code to check whether an actual argument matches the specified pattern. The actual argument is known to be in root normal form at this point. It is tested whether the symbol matches the symbol of the pattern. If this test fails, the rule does not match, all sub-nodes are popped off the stack and the execution proceeds with the next rule alternative. If the pattern is not a basic value the sub-arguments are pushed and matched.

```
match_arg :: tree -> bindings -> states -> asp -> label -> a_src -> arity ->
                                                (abc_assembler,bindings,states,asp)
match_arg (NODE anns id (INTval i) [ ]) binds states asp next a_src arity
 = (code ,binds, states, asp)
   where  code
            = typecheck                                    ++
              [  EqI_a       i a_src                    ]  ++
              escape_false (asp-arity) next
          typecheck
            = []                                             , if member anns IsINT
            = [  Eq_desc     int_symbolid a_src         ]  ++
              escape_false (asp-arity) next                  , otherwise
match_arg (NODE annots id symbol args) binds states asp next a_src arity
 = (code, bindings", states", asp")
   where  code
                     = [  Eq_desc_arity   (symbol_id symbol) n_arity a_src]   ++
                        escape_false (asp-arity) next                          ++
                        pushargs a_src n_arity                                 ++
                        match_sub_args
          bindings'  = bind_args args asp' binds
          states'    = record_args args states
          asp'       = asp + n_arity
          n_arity    = # args
          (match_sub_args,bindings",states",asp")
                     = match_args args bindings' states' asp' next asp' arity
```

escape_false n next generates code to pop the sub-arguments from the stack and a jump to the next alternative if the match failed.

```
escape_false :: nr_args -> label -> abc_assembler
escape_false nr_args label
 = [  Jmp_false  label                                 ]  , if nr_args = 0
 = [  Br_true    2                                     ,
      Pop_a      nr_args                               ,
      Jmp        label                                 ]  , otherwise
```

Rewriting according to a rule alternative with contractum

The contractum is built in a number of phases. First the shared nodes defined by the node-definitions are constructed. Then the root node is handled.

```
contractum   graph -> bindings -> states  > asp  > abc_assembler
contractum (top  node_defs) bindings states asp
 = build_shared_nodes ++ root top bindings' states' asp'
   where  (build_shared_nodes,bindings ,states',asp')
                = shared_nodes node_defs (bind top_id 0 bindings) states asp
          NODE annots top_id symbol args     =  top
```

The shared nodes laying on a cycle are created first. This enables the use of their node-id's when the cycle nodes must be constructed Then the nodes are filled from top to bottom.

```
shared_nodes   [tree] -> bindings -> states ->asp->(abc_assembler,bindings,states,asp)
shared_nodes node_defs bindings states asp
 = (code, bindings", states", asp")
   where  code    =  create_cycle_nodes ++ fill_nodes_code
          (create_cycle_nodes,bindings',states',asp')
                  =  cycle_nodes node_defs bindings states asp
          (fill_nodes_code,bindings ',states",asp")
                  =  fill_nodes node_defs bindings' states' asp'

cycle_nodes   [tree] -> bindings -> states -> asp -> (abc_assembler,bindings,states,asp)
cycle_nodes [ ] bindings states asp
 = ([ ],bindings,states,asp)
cycle_nodes (NODE annots nodeid sym args rest) bindings states asp
 = (code, bindings", states' , asp ')          , if member annots OnACycle
 = cycle_nodes rest bindings states asp    , otherwise
   where  code       = [  Create      ++   code_rest
          bindings'  =  bind nodeid asp' bindings
          states'    =  record nodeid Created states
          asp        =  asp + 1
          (code_rest,bindings",states",asp") = cycle_nodes rest bindings' states' asp
```

The function fill_nodes takes care of the filling of the shared nodes in the order they are specified If necessary, a node is created

```
fill_nodes   [tree] -> bindings -> states -> asp -> (abc_assembler,bindings,states,asp)
fill_nodes [ ] bindings states asp
 = ([ ],bindings,states,asp)
fill_nodes (node  rest) bindings states asp
 = (code, bindings", states", asp")
   where  code    =  creation    ++     fill_code     ++     fill_rest_code
          creation
                  =  [ ]                                        , if previously_defined
                  =  [ Create                               ]   , otherwise
          bindings'
                  =  bindings                                   , if previously_defined
                  =  bind nodeid asp' bindings                  , otherwise
```

```
asp'
        = asp                                  , if previously_defined
        = asp + 1                              , otherwise
a_src   = asp' - bindings' nodeid
(fill_code,states')   = fill_node node bindings' states asp' a_src
(fill_rest_code,bindings",states",asp") = fill_nodes rest bindings' states' asp'
NODE annots nodeid sym args          = node
previously_defined                   = states nodeid = Created
```

The function fill_node fills an existing node with the specified value. It is used to fill shared nodes as well as arguments. Filling a node with a basic value is simple. Otherwise, code to construct the arguments is generated before the fill instruction. When a strictness annotation is present in the node, the function fill_strict is called to fill the node with a root normal form.

```
fill_node :: tree -> bindings -> states -> asp -> a_src -> (abc_assembler,states)
fill_node (NODE annots id (INTval i) [ ]) bindings states asp a_src
 = (code, states')
   where  code    = [  FillI      i a_src                                        ]
          states' = record id InRnf states
fill_node (NODE ans nid (Function sid) args) binds state asp asrc
 = fill_strict (NODE ans nid (Function sid) args) binds state asp asrc, member ans Strict
 = (code, states")                                          , otherwise
   where  code    = build_args_code ++ [ Fill sid arity (node_label sid) (asrc+arity)   ]
          (build_args_code,states")  = build_args args binds states' asp
          states' = record nid Unknown state
          arity   = # args
fill_node (NODE annots nid (Constructor sid) args) binds states asp asrc
 = (code, states")
   where  code    = build_args_code ++ [  Fill    sid arity rnf_lab (asrc + arity)      ]
          (build_args_code,states")  = build_args args binds states' asp
          states' = record nid InRnf states
          arity   = # args
```

A node is filled with a root normal form by constructing the appropriated stack frame and calling the first alternative of the specified function. Using this entry, instead of the apply entry, results in a bit more code; the code for the evaluation of strict arguments is duplicated. However, the code produced is more efficient; when a strict argument is known to be in root normal form no attempt to reduce it, is necessary.

```
fill_strict :: tree -> bindings -> states -> asp -> a_src -> (abc_assembler,states)
fill_strict (NODE anns nodeid (Function sid) args) bindings states asp 0
 = (code, new_states)
   where  code           = build_args_code ++   [   Jsr    (reduction_label sid)       ]
          new_states     = record nodeid InRnf states'
          (build_args_code,states')  = build_args args bindings states asp
```

```
fill_strict (NODE annot id (Function sid) args) bindings states asp asrc
 = (code, states)
   where  code
              = [ Push_a    asrc                        ] ++
                fill_code                                 ++
                [ Pop_a  1                              ]
          (fill_code,states)
              = fill_strict (NODE annot id (Function sid) args) bindings states (asp+1) 0
fill_strict node bindings states asp a_src
 = fill_node node bindings states asp a_src
```

Building arguments

Arguments are constructed by Build_args, the first argument specified is built last. So, it will be on the top of the A-stack as required.

```
build_args :: arg's -> bindings -> states -> asp ->(abc_assembler,states)
build_args [ ] bindings states asp
 = ([ ],states)
build_args (arg:args) bindings states asp
 = (code, states")
   where  code                   = args_code ++ arg_code
          (args_code,states')    = build_args args bindings states asp
          (arg_code,states")     = build_arg arg bindings states' (asp+#args)

build_arg :: arg -> bindings -> states -> asp -> (abc_assembler,states)
build_arg (NodeId annots nodeid) bindings states asp
 = (push ++ evaluation   , new_states  )  , if reduction_needed
 = (push                 , states      )  , otherwise
   where  new_states             = record nodeid InRnf states
          push                   = [  Push_a    (asp - bindings nodeid)  ]
          evaluation             = [  Jsr_eval                           ]
          reduction_needed  =  member annots Strict & ~(states nodeid=InRnf)
build_arg (Term annots node) binds states asp
 = (creation ++ eval_code   , strict_states )  , if member annots Strict
 = (creation ++ fill_code   , fill_states   )  , otherwise
   where  creation               = [  Create                             ]
          (eval_code,strict_states) = fill_strict node binds states (asp+1) 0
          (fill_code,fill_states)   = fill_node node binds states (asp+1) 0
```

Build contractum and overwrite root

Finally, the root of the contractum is constructed. If it is a constructor a root normal form is reached. After building the result a rtn instruction is generated to return to the initiator of the reduction. If the node contains a function symbol, the stack frame is adapted to the one required by the new function. A new stack frame for a function is created by building it on top of the current stack frame and than copying it to the bottom. The new function is invoked by a jump instruction. In this way tail recursion is removed for free!

```
root :: tree -> bindings -> states -> asp -> abc_assembler
root (NODE annots id (INTval i) [ ]) bindings states asp
 = pop_args asp ++
   [  FillI       i 0                                        ,
      Rtn                                                    ]
root (NODE annots nid (Constructor sid) args) bindings states asp
 = build_args_code    ++
   [  Fill        sid arity rnf_lab (asp+arity)              ] ++
   pop_args asp       ++
   [  Rtn                                                    ]
   where  (build_args_code,states')   = build_args args bindings states asp
          arity  =  # args
root (NODE annots nid (Function sid) args) bindings states asp
 = build_args_code          ++
   clean_up (# args) asp    ++
   [  Jmp       (reduction_label sid)                        ]
   where  (build_args_code,states')   = build_args args bindings states asp
```

Rewriting according to a redirection

A redirection is implemented by coping the root of the graph indicated to the redex. If it is known at compile time that this graph is in rnf it is just copied, otherwise it is reduced first in order to prevent the duplication of work.

```
redirection :: annots -> node_id -> bindings -> states -> asp -> abc_assembler
redirection annots nodeid bindings states asp
 = [  Fill_a      newroot asp                                ,
      Pop_a       asp                                        ,
      Rtn                                                    ]  , if states nodeid = InRnf
 = update_astack newroot (asp-1)   ++
   pop_args (asp-1)                ++
   [  Jsr_eval                                               ,
      Fill_a      0 1                                        ,
      Pop_a       1                                          ,
      Rtn                                                    ]  , otherwise
   where  newroot    =  asp - (bindings nodeid)
```

The additional rewrite alternative entry

A last entry is generated when the function has arguments; a rule alternative without arguments will always match, hence an additional entry is not needed.

```
fill_with_rnf :: symbolid -> arity -> num -> abc_assembler
fill_with_rnf symbol_id arity nr_alts
 = []                                                    , if arity = 0
 = [  Label      (alt_label symbol_id (nr_alts+1))     ,
      Fill       symbol_id arity rnf_lab arity          ,
      Rtn                                               ]  , otherwise

gen_type_error :: symbolid -> arity -> num -> abc_assembler
gen_type_error symbol_id arity nr_alts
 = []                                                    , if arity = 0
 = [  Label      (alt_label symbol_id (nr_alts+1))     ,
      Jmp        type_error                             ]  , otherwise
```

5.5 The run-time system

The compilation scheme presented here produces code for each rewrite rule to reduce a graph to rnf. The run-time system contains the routine init_graph to build the start node and initiates its reduction to normal form. The reduction to normal form and the printing of sub-graphs in normal form is done by the routine _driver. Furthermore the descriptors for basic types and some general entries are defined. The _rnf entry is used for graphs in root normal form. The _cycle entry is stored in a node currently reduced. In this way it is detected that such a node is revisited, the functional strategy cannot find a rnf, hence an error message is generated. The type_error entry is used when none of the rule alternatives is applicable.

```
[             Descriptor
"INT"         "_rnf" 0 "integer"                 ,   || Must be first descriptor!!
              Descriptor
"BOOL"        "_rnf" 0 "boolean"                 ,   || Must be second descriptor!!
              Jmp              "init_graph"      ,   || Must be first instruction!!
              Label                                  || The initiator of graph reduction
"init_graph", Create                             ,   || Create and fill the start node
              Fill             "Start" 0 "n_Start" 0 ,
              Jsr              "_driver"         ,   || Print the nf of start node
              Print            "\n"              ,   || Print a newline
              Halt                               ,   || Finished!

              Label                                  || The global print driving routine
"_driver",    PushI            0                 ,   || Closing bracket count
              Label                                  || Label for tail recursion
"_print",     Jsr_eval                           ,   || Reduce top node to rnf
              Get_node_arity   0                 ,   || Get number of arguments in node
              EqI_b 0 0                          ,   || No arguments ?
              Jmp_false        "_args"           ,
              Label                                   || Print last argument
"_print_last", Print_symbol  0                      , || Do it
              Pop_a          1                      , || Remove node
              Pop_b          1                      , || Remove arity
              Label                                   || Print the closing brackets
"_brackets",  EqI_b          0 0                    , || Bracket count equal 0 ?
              Jmp_true       "_exit"                , || Yes; finished
              Print          ")"                    , || No; print bracket
              DecI                                  , || Decrement bracket count
              Jmp            "_brackets"            , || Next bracket
              Label                                   || Finished with this node.
"_exit",      Pop_b          1                      , || Remove bracket count
              Rtn                                   ,
              Label                                   || Start the printing of arguments
"_args",      Print          "("                    , || An opening bracket
              Print_symbol   0                      , || The Symbol of the node
              Get_desc_arity 0                      , || Arity corresponding to symbol
              Repl_args_b                           , || Replace node by its arguments
              Pop_b          1                      , || Pop descriptor arity
              Label                                   || Loop to print arguments
"_arg_loop",  Print          " "                    , || Space between elements
              EqI_b          1 0                    , || Last argument ?
              Jmp_false      "_next_arg"            ,
              Pop_b          1                      , || Remove arg counter
              IncI                                  , || Increment bracket counter
              Jmp            "_print"               , || Optimized tail recursion
              Label                                   || Print an argument; not the last one
"_next_arg",  Jsr            "_driver"              , || Recursion to print argument
              DecI                                  , || Decrarg count; driver removes arg
              Jmp            "_arg_loop"            , || Next argument

              Label                                   || This graph is already in rnf;
"_rnf",       Rtn                                   , || Return immediately
              Label                                   || Functional strategy finds no rnf!
"_cycle",     Print  "Cyclic computation! Reduction interrupted",
              Halt                                  , || Stop the reduction
              Label                                   || No rule alternatives is applicable!
"type_error", Print  "Type error! Reduction interrupted",
              Halt                                  ] || Stop the reduction
```

5.6 Curried functions

The generic AP rule converts a curried symbol into its ordinary form as soon as all its arguments are available. Each curried symbol is a constructor since no rewrite rule can be applied, hence its context is _rnf. Each AP node provides one additional argument. When the number of arguments required by the symbol is collected, a stack frame according to the apply entry is constructed and execution continues on that entry. Otherwise, the symbol and the arguments collected are stored in the node containing the AP.

```
[          Descriptor
"AP"       "a_AP" 2 "AP"                ,
           Label
"n_AP",    Set_entry       "_cycle" 0   ,  || Mark node as being reduced
           Push_args       0 2 2        ,  || Push arguments of AP
           Label
"a_AP",    Jsr_eval                     ,  || Evaluate first argument; the function
           Label
"AP1",     Get_node_arity  0            ,  || Number of args in node
           Get_desc_arity  0            ,  || Number of args for rule
           SubI                         ,  || Number of arguments needed
           EqI_b           1 0          ,  || More than 1 argument needed ?
           Jmp_false       "args_needed" ,
           Push_ap_entry   0            ,  || Push apply entry of function on C-stack
           Get_desc_arity  0            ,  || Push arity
           SubI                         ,  || Compute number of arguments in node
           Push_b          0            ,
           Repl_args_b                  ,  || Construct top of stack frame
           Pop_b           2            ,  || Clean up B-stack
           Rtn                          ,  || to apply entry pushed above
           Label
"args_needed"                           ,
           Push_a          1            ,  || Reference to curried function
           Add_args        1 1 3        ,  || Add argument and overwrite AP node
           Pop_a           2            ,  || Clean up stacks
           Pop_b           1            ,
           Rtn                          ]  || Rnf reached
```

This is illustrated by the following example:

```
Start -> AP Inc 2   ;
Inc   -> AP Plus 1  ;
Plus  -> +I         ;
```

Several states of the reduction are depicted below.

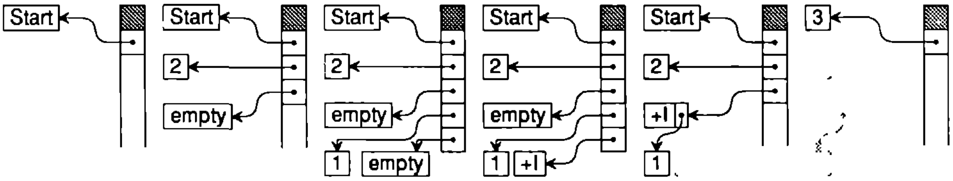


Fig 5.6.a **b** **c** **d** **e** **f**

a: Initial state. The reduction starts at the node entry of Start.
b: Creation of the arguments of AP. Second argument is built. First argument is strict.
c: Reduction according to Inc. Second argument is created, first argument is strict.
d: Reduction according to Plus. The redex node is filled with a rnf.
e: +I has not enough arguments to become a function. So, the node is filled with the rnf.
f: After reduction according to +I the top node is replaced by its rnf.

5.7 Optimizations

The code generated by the compilation scheme presented in section 5.4, can be improved at many points. The most important improvement is to use the B-stack instead of nodes to pass basic values between functions. The use of the B-stack is described informally and illustrated with an example. Afterwards some other optimizations are mentioned.

The extended compilation scheme using the B-stack

Basic values are manipulated always on the B-stack in the ABC-machine. Every computation involving basic values requires the transportation of the values to the B-stack and the shipment of the result back to a node in the graph store. When the result is used again as argument in another computation there is much data transportation. To reduce the unnecessary movement of data, the compilation scheme must be changed such that basic values stay on the B-stack as much as possible.

To achieve this, the calling conventions for the rewrite alternatives are changed: strict arguments of a basic type are passed on the B-stack and the result of a reduction is left on the B-stack when it is of a basic type. The calling conventions for the node entry and the apply entry remain unchanged. The code corresponding to the apply entry takes care of the transport of basic values between the graph and the B-stack for this function. After the reduction of a strict argument of a basic type it is transported to the B-stack. When the result of the function is of a basic type the first rule alternative entry is called as a subroutine. The rnf produced on the B stack is transported to the node containing the redex. The complexity of the code generation is increased significantly by this new calling convention. Arguments and results must be moved to the desired place at every occurrence. This is not difficult, but involves an elaborated case analysis.

An example of the use of the B-stack

To show the changes resulting from the new calling conventions the ABC-code for the Length example of section 5.2 is shown.

```
:: Length ! INT ! (List x)    -> INT                 ;
   Length n   (Cons a b)      -> Length (+I n 1) b   |
   Length n   Nil             -> n                   ;
```

Both arguments are strict. The first argument is also a basic value. So, it will be passed on the B-stack to the rule alternatives. The result will also be passed on the B-stack by the rule alternatives. The rewrite entry for +I also expects its arguments and leaves its result on the B-stack.

```
[             Descriptor
"Length"      "a_Length" 2 "Length"       ,  || The generated descriptor
              Label                          || The node entry: preparing the elements
"n_Length",Set_entry      "_cycle" 0      ,  || Mark node to detect cyclic computations
              Push_args   0 2 2           ,  || Push the arguments
              Label                          || The apply entry: reduce strict arguments
"a_Length",Jsr_eval                       ,  || Reduce first argument to rnf.
              PushI_a     0               ,  || Copy first argument to the B-stack.
              Pop_a       1               ,  || Pop first argument from A-stack.
              Jsr_eval                    ,  || Reduce second argument to rnf.
              Pop_a       1               ,  || Pop duplicated second argument.
              Jsr         "Length1"       ,  || Initiate rewriting by alternatives.
              FillI_b     0 0             ,  || Fill node with result of the reduction.
              Pop_b       1               ,  || Pop result of the reduction.
              Rtn                         ,  || Done; node is in rnf.
              Label                          || Entry for first rule alternative.
"Length1",    Eq_desc_arity "Cons" 2 0    ,  || Match arg 2.
              Jmp_false   "Length2"       ,  || Goto next alternative if match fails.
              Push_args   0 2 2           ,  || Push sub-arguments.
              Push_a      1               ,  || Rewrite according to alternative 1; Push b
              Jsr_eval                    ,  || Reduce it.
              PushI       1               ,  || Second argument for +I.
              Push_b      1               ,  || First argument for +I.
              Jsr         "+I1"           ,  || Reduction of +I n 1.
              Update_a    0 3             ,  || Adapt A-stack frame for call of Length.
              Update_b    0 1             ,  || Adapt B-stack frame for call of Length.
              Pop_a       3               ,  || Remove old arguments from A-stack.
              Pop_b       1               ,  || Remove old arguments from B-stack.
              Jmp         "Length1"       ,  || Continue with alternative 1 of Length.
              Label                          || Entry for second rule alternative.
"Length2",    Eq_desc_arity "Nil" 0 0     ,  || Match argument.
              Jmp_false   "Length3"       ,  || Continue with alternative 3 if match fails.
```

```
            Pop_a       1              ,  || Remove arg.
            Rtn                        ,  || Rnf reached!
            Label                         || Generated entry for additional alternative.
"Length3",  Jmp         "type_error"   ]  || This must be a type error!
```

In figure 5.7, snapshots of the reduction process corresponding to the pictures in Fig 5.5. are shown.

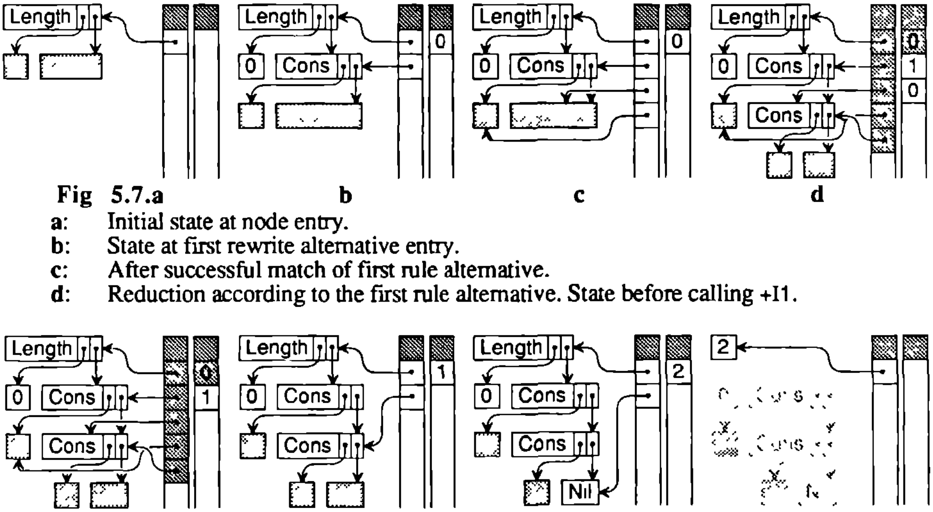


Fig 5.7.a b c d

a: Initial state at node entry.
b: State at first rewrite alternative entry.
c: After successful match of first rule alternative.
d: Reduction according to the first rule alternative. State before calling +I1.

e f g h

e: State returned by +I1.
f: State before calling Length1 recursively.
g: State before last call of Length1.
h: State of ABC-machine after the rnf is reached.

This example shows the successful use of the B-stack, very few movement of data between the graph and B-stack is necessary. The next example shows that one is not always that lucky.

```
::  I!x           -> x           ;
    I x           -> x           ;

::  F !INT !INT   -> INT         ;
    F a b         -> +I (I a) b  ;
```

Both arguments and the result of function F will be passed on the B-stack by the rule alternative entry. The identity function expects its argument and leaves its result on the A-stack. So, a has to be put in a node to pass it to I and the result must be extracted of a node to pass it to +I.

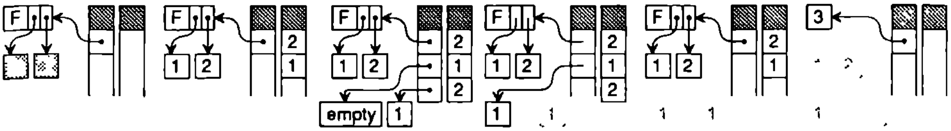


Fig 5.8.a b c d e f

a: Machine state at node entry of rule F.
b: State at rewrite alternative entry for F.
c: Second argument of +I copied to the B-stack. Initiating the reduction of first strict argument.
d: The rnf returned by I.
e: The stack frame to call +I1.
f: Finally, the rnf is copied to the redex by the apply entry of F.

Other optimizations

The code generated by the schemes above is more efficient than the scheme in 5.4, but can be improved further. Here we mention some other optimizations possible.

Improve the performance of predefined functions (delta rules) by hand-coding them in ABC-assembler.

The use of in-line code substitution for very short functions. For instance, the AddI instruction instead of a Jsr +I1.

Use information of predefined functions to improve the code generated by the compilation schemes. For instance, the predefined conditional function:

```
:  IF !BOOL  type  type  -> type  ;
   IF True   then  else  -> then  |
   IF False  then  else  -> else  ;
```

Obviously, IF is strict in its first argument. Its second and third argument are never both needed. In a strict context building the then and else part can be delayed until the condition is evaluated. Then it is clear which one is needed, only that one has to be built.

In the present compilation schemes the matching of each rule alternative starts from scratch. However, the use of information from the previous alternative and type information can speed up the matching process considerably. So, it is better to use a finite state machine for the matching of the rule alternatives instead of starting all over for each alternative.

Constant graphs which occur in the rhs of rewrite rules need not to be built each time the rule is used. These graphs can be created once. The node-id's of the root of these graphs can be used instead of building a new one at each occurrence.

Use a scratch node for nodes which will be used once for a very short time. This occurs when a function delivers its result in the graph which is required on the B-stack; the result is stored in the node, moved to the B-stack and the node becomes garbage immediately.

Efficiency gain of the optimizations

To analyse the effect of the mentioned optimizations a measurement of the performance of the well known nfib function is presented. The integer delivered by the nfib function is the number of function calls done. The nfib-number is obtained by dividing this number by the execution time. So, the nfib-number is the number of function calls per second.

```
::  Nfib !INT   -> INT                                     ;
    Nfib 0      -> 1                                       |
    Nfib 1      -> 1                                       |
    Nfib n      -> ++I (+I (NFib (--I n)) (Nfib (-I n 2)))  ;
```

The table below gives an idea about the gain in efficiency obtained by the optimizations. However this example is not representative for all programs. The nfib number is extremely sensitive for the optimizations discussed.

These measurements where done on a SUN 3/280 under SunOS 4.0.3, using the ABC-machine implementation made by Gé Weijers [Weijers 90]. Measurements are accurate within 10%.

Nfib-number	*Conditions*
27,000	Code generation according to the scheme presented here. No strictness information nor type information is provided. At run-time a type check on the argument of Nfib is performed.
35,000	Using type and strictness information for the delta rules only. In-line code substitution is used for these delta rules. In the rest of the bench-marks these delta rules are used.
43,000	Using type information at compile time; the type check on the argument of the nfib function is dropped.
70,000	Using only the strictness information of the nfib function. No type information is supplied. So, the type of the argument will be checked at run-time.
90,000	Type and strictness information are supplied to the basic translation scheme.
280,000	Using type and strictness information in the extended translation scheme. The integers are passed on the B-stack but no in-line code is used for the delta rules.
415,000	Using type and strictness information in the extended translation scheme. The integers are passed on the B-stack and in-line code is used for the delta rules.

495,000	Result of the same ABC-code, but using a better register allocation in the ABC-compiler [Groningen 90]. The ABC-code resulting from an nfib-function definition with a conditional is more efficient. The corresponding Nfib-number on the Sun is 571,000 on a Mac IIfx this nfib-number is 1,009,000!
2,000	Same code interpreted by the PABC simulator [Nöcker 89].
300,000	A similar nfib function written in C [Kernighan 78].
1,200	The nfib function written in Miranda; executed by the interpreter.
1	ABC-specification executed by the Miranda interpreter
10	ABC-specification without superfluous checks in the micro-instructions, executed by the Miranda interpreter.

The code for Nfib in ABC-assembly using the extended compilation scheme becomes:

```
[           Descriptor
 "Nfib"     "a_Nfib" 1 "Nfib"            , || The generated descriptor
            Label                          || The node entry. prepare arguments
"n_Nfib",   Set-entry    "_cycle" 0      , || Mark node to detect cyclic computations
            Push_args    0 1 1           , || Push argument on A-stack
            Label                          || The apply entry.
"a_Nfib",   Jsr_eval                     , || Reduce strict argument
            PushI_a      0               , || Copy argument to B-stack
            Pop_a        1               , || Remove argument from A-stack
            Jsr          "Nfib1"         , || Initiate calculation on B-stack
            FillI_b      0 0             , || Copy result from B-stack to node
            Pop_b        1               , || Remove result from B-stack
            Rtn                          , || Rnf reached
            Label
"Nfib1",    EqI_b        0 0             , || match rule 1; is argument 0?
            Jmp_false    "Nfib2"         , || No; continue with second rule alternative
            Pop_b        1               , || Yes; remove arg
            PushI        1               , || Store result
            Rtn                          , || Done
            Label
"Nfib2",    EqI_b        1 0             , || Match rule 2; is argument equal to 1?
            Jmp_false    "Nfib3"         , || No; continue with rule alternative 3
            Pop_b        1               , || Yes; remove argument
            PushI        1               , || Push result
            Rtn                          , || Done
            Label                          || Rule alternative 3; no matching required
"Nfib3",    PushI        2              ,  || Compute -I n 2
            Push_b       1              ,  || Push n
            SubI                        ,  || In-line code substitution for -I
            Jsr          "Nfib1"        ,  || Compute Nfib (-I n 2)
            Push_b       1              ,  || Compute --I n; push argument
            DecI                        ,  || In-line code substitution for --I
            Jsr          "Nfib1"        ,  || Compute Nfib (--I n)
            AddI                        ,  || In-line code substitution for +I
            Update_b     0 1            ,  || Replace argument by result
            Pop_b        1              ,  || Remove copied result
            IncI                        ,  || In-line code substitution for ++I
            Rtn                         ]  || Done
```

5.8 Discussion

This chapter presents an abstract compilation scheme to translate Clean to ABC-assembler. This is an example of a non-trivial translation containing context sensitive decisions. Due to the nature of the translation to be performed the specification is rather elaborate, but it remains very readable.

It appears to be valuable that the specification is executable; the generated code can be observed and executed. Execution serves as a test for the specification, due to its nature it is very hard to verify its correctness. Using this prototype it is easy to glance at the code generated according to slightly different schemes; it appears to be easy to make small changes in the compilation schemes. Combining this prototype with the prototype of the ABC-machine presented in chapter 3 it is possible to execute the Clean programs. Important observations of the dynamically behaviour can and have be obtained in this way. Furthermore there are of course the usual advantages: the consistency can be partially checked by the Miranda compiler and the semantics of the description language are clear.

The specification is used as guide-line for the construction of the corresponding product [Smetsers 89]. This compiler uses the B-stack in the way described here and incorporates the first three other optimizations mentioned above as well as many other optimizations. Although it is in principle possible to describe each and every optimization in a specification this is not done. The specification obtained would be very large and, hence, almost as difficult to read as the actual implementation. The specification presented here shows clearly the kind of code generated and enables the informal description of many optimizations.

Chapter 6
Artificial Neural Networks

The popularity of artificial neural networks is rapidly increasing. These networks appear to be able to solve relatively easy some problems that are hard to solve with conventional algorithms. Especially pattern recognition tasks are performed very well by some networks. In order to detect which aspects of neural networks are responsible for this success, it is necessary to have a good understanding of the paradigm used in neural network computations. We will show that functional languages can be used to obtain a uniform description of various artificial neural networks. Such a uniform high-level description will make it easier to spot the essential differences and similarities among the neural network models used.

There are many possible descriptions of a biological neural network. Each description shows some aspects of the network. Some of the possible descriptions are: morphological, chemical (the description of the chemical reactions and matters involved, e.g. neurotransmitters), biological (description of the neurons as biological cells), physical (spike generation and travelling over the axons, or system description of large ensembles of neurons) and psychological (description of complicated behaviour). The missing item in this list is a description of the type of computations performed and algorithms used by a neural network. The nature of these computations is largely unknown.

One of the ways to explore the computational models used in the brain is to make a study of artificial neural networks. These artificial networks are models inspired by the brain. One usually starts with very simple models that can perform only a very limited set of tasks. Other features, increasing the complexity, are introduced step by step when the current model cannot execute the next task. More than 50 networks are proposed in the literature [Hecht-Nielsen 87b, 88].

In this chapter a specification of some important artificial neural networks in a functional programming language is given. First, a set of description tools is defined. Then, each specific network is described with these primitives. For the description of artificial networks it is very convenient that the descriptions are executable. Due to the used nonlinear functions and stochastic terms a complete analytical study of the properties of these systems is very hard or impossible.

These descriptions show that the processing elements used in these networks are very similar. The use of these elements distinguishes the various networks. To indicate the huge simplifications made in artificial networks with respect to real networks a rough sketch of biological neural networks is given.

Biological neural networks

The human brain consists of a very huge number (estimations range from 10^{10} to 10^{13}) of neurons (brain cells), each one receives input from other neurons (typical about 10^{4}) [Schadé 84]. Based on this input a short pulse on the output, called a spike, may be generated. This spike travels over the output axon and serves as input for other neurons. A spike is a pulse-shaped change of the potential over the axon membrane.

Nerve conduction is studied since the late 1940's. The shape and conduction speed along the axon of a spike are related to the observed changes in membrane permeability by the Hodgkin Huxley equations [Hodgkin 52]. The model does not explain why the membrane permeability for various ions changes; it describes just the consequences. These equations can be found in many textbooks, e.g. [Hobbie 78].

When a spike reaches the connection of the output axon with some input axon it changes the state of the corresponding neuron. In most situations chemical substances, called neurotransmitters, are released in the synaptic junction in quantized packets. These neurotransmitters change the interior potential of the post synaptic neuron, it becomes either more positive or more negative depending on the neuro transmitter released. If the potential becomes high enough, above the threshold value, a spike is generated on its output axon. Properties of neurotransmitters are studied since long, e.g. [Koopman 60].

There is morphological evidence that the number of synapses in the brain increases rapidly with the age. It is believed that this is one of the mechanisms to let the brain learn.

6.1 Description tools for artificial neural networks

Artificial neural networks are a crude simplification of biological networks. These models are most suited for relatively simple tasks, like sensory and motor functions. Each artificial neuron is a simple processing element which receives input from a fixed set of neurons. The output value is not continuously computed in the massive parallel way of the brain, but in discrete time steps. Artificial neurons do not communicate by spikes, but by a sequence of values representing neuron activity. A single scalar number represents the strength and state of a synapse. The input value is multiplied by this weight to obtain the contribution of this input to the potential. A learning procedure adjusts these weights to let the network perform the desired task. The number of processing elements used ranges from one to several thousands.

This chapter presents a set of description tools for artificial neural networks and uses them to specify some well-known networks. This approach does not always yield the most direct and compact description of a given network, but makes the similarities and distinctions between the various networks better visible. Here it is assumed that the network has a layered structure of uniform processing elements. Since the networks described here can be cascaded also networks with layers of different nature can be captured by this description method. The tools developed here are also used to describe non-layered networks [Koopman 90, Rutten 90].

Artificial neural networks are characterised by

- the values processed;
- the processing elements used;
- the topology of the network (interconnection of processing elements);
- the synchronous or asynchronous character of the computations; and
- the way the processing elements are tuned to let the network perform some task. Elements are either completely predetermined or some parameters are tuned in a learning phase.

First, a set of description tools is developed in a brief discussion of these characteristics. Then, some neural networks are described using these tools.

The values processed

The values on the axons are numbers representing neuron activity. There are two types of domains used; either binary or continuous.

A binary activity quantity has one of two possible values; these values represent firing at maximum rate and quiet. The numerical values are usually 0 and 1, or -1 and 1. The latter variant has some advantages since most synapses compute a weighted sum of the input values, but every weighted sum of zeros yields 0. Moreover, in many applications it is convenient that *on* and *off* of a neuron are in some sense symmetrical.

A continuous activity quantity can have any value within some limits as 0 and 1, or -1 and 1. These bounds have the same meaning as for binary values, but also intermediate states of neuron activity can be modelled.

Our description uses the type value for neuron activity, ordinary numbers are used to represent it. When appropriate, values are indicated as input, output or target value.

```
value        ==  num
input_value  ==  value
output_value ==  value
target_value ==  value
```

A collection of simultaneously occurring values is modelled by a list of values called state.

```
state        ==  [value]
input_state  ==  state
output_state ==  state
target_state ==  state
```

Neuron characteristics

In the simplest model a neuron takes an input state and yields an output value determined by this state. In more complicated models there may be additional parameters such as a noise input or a history term representing some dynamic properties. Most of the neural network models found in the literature do not need these extensions. Therefore, we will model our neurons by:

```
neuron       ==  input_state -> output_value
```

Each neuron consists of a synapse which yields an internal potential, and a generator which produces the output value based on this potential. This element is introduced by McCulloch and Pitts [McCulloch 43] and has become well-known through the work of Hopfield [Hopfield 82]. Artificial neural networks usually employ this type of elements. This model forms an abstraction of biological neurons.

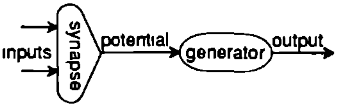


Fig 6.1 The McCulloch-Pitts model of a neuron.

The internal potential of an element is fully determined by the current input.

The synapse

A synapse is a device (function) that computes the internal potential based on the input state. Usually, a weighted sum of the input values is used. The only difference between various neurons is the set of weights used to compute the potential. Learning is modelled by changing these weights. For these reasons the weights are passed as arguments to the synaptic function and to the elements.

```
weights    ==   [num]
potential  ==   num
synapse    ==   weights -> input_state -> potential
```

Some well-known synaptic functions are:

```
wsum weights inputs = sum [weights!i * inputs!i | i <- [0..#weights -1]]
```
[1]

```
neocognitron_synapse weights input
 = (1+excitation) / (1+inhibition) - 1
   where  excitation  =   sum [ weights ! i * input ! i | i <- [0..#weights-1]; weights ! i > 0 ]
          inhibition  = - sum [ weights ! i * input ! i | i <- [0..#weights-1]; weights ! i < 0 ]
```

When one views the sequence of weights and the input state as vectors, wsum just computes the inner product (scalar product) of these vectors.

The generator

The generator function yields the output value determined by the potential.

```
generator  ==   potential -> output_value
```

[1] In this chapter ZF-expressions are used to specify matrix and vector like operations, instead of the equivalent map's, zip's and transposes. The given definition resembles the mathematical expression

$$\sum_{i=0}^{N-1} \text{weigths}_i \times \text{inputs}_i$$

more than the equivalent

```
sum (map2 (*) weights inputs)
```

When an index is used at several places in an expression we generate the sequence of indices rather than the sequence of elements chosen. Using the generation of list elements instead of the generation of indices, the definition above would become:

```
sum [weight * input | (weight,input) <- zip2 weights inputs]
```

This generator is generally a nonlinear function of the potential. Some models involve a stochastic term in the generation of the output value. This is adequately modelled by adding a noise term to the potential. The most often used generator functions are a hard limiter for a binary output and a sigmoid or pseudo linear function for continuous output values. Some well-known generator functions are:

```
hard_limit v                                || binary output
 = high, if v > threshold
 = low , otherwise
threshold = (high + low) / 2
```

The optimum threshold value depends on the network model used and the specific problem at hand.

```
sgn v                                       || binary output
 =  1, if v > 0
 = -1, otherwise
```

```
clip v                                      || continuous output, pseudo linear
 = low   , if v <= low
 = v     , if low < v <= high
 = high  , if high < v
```

```
sigmoid v = (high-low) / (1+exp (-s*v)) + low  || continuous output
```

s is a small positive constant determining the 'sharpness' of the function. For very large values of the sharpness, the sigmoid function resembles the step function.

The bounds high and low are separately defined for each specific network.

In many models the input of the generator is the difference of the potential and a threshold value, instead of the potential itself. Whenever the potential is determined by a weighted sum, which is by far the most used synaptic function, a connection to a fixed valued input can replace the threshold. For a generator function, f, the output is determined by

```
output     = f (potential - threshold)
potential  = wsum weights input_state
```

This is equivalent to

```
output     = f potential'
potential' = wsum (-threshold: weights) (1: input_state)
```

The latter formulation is more convenient since no individual threshold values for the elements have to be maintained. Another benefit of this formulation is that the threshold can be optimized by the learning algorithm used for the weights.

Processing elements

The proper combination of synaptic function and generator function specifies the processing elements. Some well-known processing elements are:

```
element      ==   weights -> neuron

adaline           weights   =  sgn        .wsum weights   || binary output values
perceptron_b      weights   =  hard_limit .wsum weights   || binary output values
perceptron_c      weights   =  sigmoid    .wsum weights   || continuous output values
hopfield_b        weights   =  hard_limit .wsum weights   || binary output values
hopfield_c        weights   =  clip       .wsum weights   || continuous output values
```

The usual choice for the bounds high and low is 1 and -1 respectively. With these values the functions sgn and hard_limit become equivalent. This makes the processing elements used in the various networks even more similar.

Neuron interconnection patterns and evaluation orders

Artificial neurons are usually arranged within layers. The layers consist of identical elements, the only difference between the various neurons is the vector weights used in the synapse. We prefer to specify a layer of neurons by the element function for all neurons and the weight vector for each element, instead of a sequence of neuron functions. It is much easier to adapt these weight vectors by a learning algorithm and to monitor their evolution than to change and monitor the evolution of a function.

```
weights_layer   ==   [weights]          || the synapse weights in a layer
weights_net     ==   [weights_layer]    || the weights in a network
net_state       ==   [state]            || the output states of all neurons in the network
```

In this description the size of the network is deduced from the sizes of the weights set determining the synapses in the network. Usually, the size of a network is not changed during its use, albeit it is possible that the learning algorithms change the size of the network dynamically. The initial network is generated by the function initial_net which is defined for each network separately. The size of the generated net is determined by the list of numbers.

```
initial_net :: [num] -> weights_net
```

To avoid computational problems resulting from cyclic dependencies, it is usually assumed that there are only unidirectional connections between two adjacent layers. The units in the first layer, called the input layer, receive the input state and yield a state to be processed by the next layer etcetera. Finally, the last layer, the output layer, produces the response of the network.

The output state of a layer of neurons is specified by a function parametrized by the element used, and the list of weight vectors for each individual element. This function takes the input state as argument and yields the corresponding output state.

```
eval_layer :: element -> weights_layer -> input_state -> output_state
eval_layer element layer input = [ element weights input | weights <- layer]
```

With a single layer of neurons only very simple networks can be built. Using a cascade of these layers, a network with better properties can be built: a multilayer feed forward network. A list of representations of layers represents a layered network. Here it is assumed that all layers contain identical elements. A network consisting of layers with different elements in the layers can be described by a cascade of the networks described here. The constant neurons, used to represent the threshold, are added to each layer. The number of constant neurons is determined in the specification of a specific network.

```
feed_forward :: element -> weights_net -> input_state -> net_state
feed_forward element weights input
 = state
   where  state
                = (input ++ constant_neurons):              || state!0 is the input state
                  [eval_layer element (weights!l) (state!l) ++ constant_neurons
                  | l <- [0..#weights -1]]

constant_neurons :: state
constant_neurons = rep nr_constant_neurons high

select_output :: net_state -> output_state
select_output state
 = output_state                                        , if nr_constant_neurons = 0
 = take (#output_state - nr_constant_neurons) output_state, otherwise
   where  output_state = last state
```

The input state is recorded in the state of the network since it is needed in several learning algorithms. Unfortunately, this additional vector in the state of the network has as consequence that the element determined by the vector at weights!l!i corresponds to state!(l+1)!i.

Some artificial neural networks do have connections within one layer, the introduction of a delay solves the computational problems due to the cyclic dependencies. The neuron uses the output produced some time steps ago as input, instead of the current output. In these feedback networks an input state is presented to the network and the output is used as new input state. This requires that the input state and output state have the same size. These networks consist usually of a single layer of elements. For feedback networks it matters whether the elements are updated synchronously or asynchronously. When the elements are evaluated asynchronously, i.e. one by one, the element that is updated as last one, 'sees' almost the new state. When the elements are synchronously evaluated all elements have the same input state. The function feed_back_syn searches a stable state in a sequence of states generated by an iteration of feed_forward passes over the initial state.

```
feed_back_syn :: element -> weights_net -> input_state -> [output_state]
feed_back_syn element weights input
 = to_limit (iterate (select_output.feed_forward element weights) input)
```

The funcion to_limit takes the leading part of a list. The list is broken when two consecutive elements are equal. It is defined as:

```
to_limit :: [*] -> [*]
to_limit  (a:b:x)
 = [a]                , if a = b
 = a : to_limt (b:x)   , otherwise
```

The function feed_back_asyn implements an asynchronously updated single layer network, the elements are updated one by one in a pseudo random order.

```
feed_back_asyn:: element -> weights_net -> input_state -> [output_state]
feed_back_asyn element [weights] input
 = to_limit (iterate (eval_one_by_one element weights) (input ++ constant_neurons))

eval_one_by_one :: element -> weights_layer -> input_state -> output_state
eval_one_by_one element w_layer input
 = eval_neurons element w_layer input [0..# w_layer -1]

eval_neurons :: element -> weights_layer -> state -> [num] -> state
eval_neurons state element w_layer [ ] = state ++ constant_neurons
eval_neurons state element w_layer indices
 = eval_neurons new_state element w_layer rest
   where  n       =  random (1993+entier (sum state)) mod # indices    || pseudo random
          rest    =  remove n indices          || remove this index from the list of neurons
          index         =  indices ! n         || index of neuron to update in this iteration
          new_state     =  update index new_value state
          new_value     =  element (w_layerlindex) state
```

There are several successful networks using a topology not covered by the class of layered networks described above, e.g. the Boltzmann machine is a randomly connected network. A sparsely connected network can sometimes be modelled adequately by a fully connected network where some of the weights are kept zero.

Learning rules

The 'knowledge' of a neural network is stored in its architecture and the weights used in the synapses. Generally, the desired weights are unknown. The function to be performed is only specified by a set of input-output pairs. The weights cannot be determined immediately, but must be deduced in some way from the samples. There are several algorithms known to obtain a reasonable set of weights by training the network. These learning rules can be classified into

- **Unsupervised learning.** These learning rules must set the weights based upon a set of sample inputs. No supervisor system is available to indicate whether the network behaves correct. This is used mainly in pattern recognition, the samples presented are the samples to be recognised later. So, sample and target are identical. The best-known unsupervised learning rule are the Hebbian-type learning ruls: the strength of a connection between two units is proportional to the correlation in activity. Activity in one unit is associated with the activity in the other unit. In this way the network *learns* to associate patterns of neural activity.
- **Supervised learning.** In these situations there is a teacher which presents sample patterns and the desired output state. The general principle is to feed a network a sample input and target. The weights in the network are adapted based on the difference between target and output state, such that the response of the network is improved when the same sample is presented to the network again. Using the target it can be detected which output units are wrong. For other units a heuristic must be used to decide whether they must be changed.

Generally, the weights are determined by adjusting a set of initial weights by adapting them to a sequence of input_states and the associated target_states. The function learn is used for this purpose.

```
learn :: weights_net -> [input_state] -> [target_state] -> weights_net
learn weights (sample:samples) (target:targets)
 = learn (adapt_weights weights sample target) samples targets
learn weights [ ] [ ]
 = weights
```

The generalized delta rule

The generalized delta rule is a supervised learning rule used, in one variant or another, in many networks. It will be proven that this rule 'works'; the weight changes prescribed by this rule reduce the error made by the network [Rumelhart 86]. The delta rule is used for networks containing McCulloch-Pitts neurons with a synapse that take a weighted sum. The proof is only valid for neurons with a nondecreasing differentiable generator function, but the delta rule is used with success for networks processing binary values. Unfortunately, the only way to check whether the network has learned to execute its task is to test it with all possible input patterns. Another problem is that it is possible to get stuck in a sub-optimal solution.

In informal terms the ordinary delta rule formulated for a single layer of binary elements, devised in 1959, can be expressed as [Widrow-Hoff]:

- Present a sample to the network and compute the corresponding output;
- If the output is equal to the target do nothing;
- If the output value is high and the target value is low, the potential (the value of the inner product) is decreased by enlarging the weights corresponding to negative inputs and decreasing the weights associated with positive inputs;
- If the output is improperly low, the potential is increased by augmenting the contribution of positive inputs and lessening the influence of negative inputs.

The formula for changing the weights associated to input i in element j, W_{ji}, is

$$\Delta W_{ji} = \alpha (target_j - output_j)\, input_i$$

The factor α, in this formula, is a small positive value determining the learning rate. This simple formula is only applicable for single layered networks, since the target values for hidden elements are generally unknown. Moreover, it is not proven that this rule yields a correctly behaving network.

To derive the *generalised delta rule*, a measure for the (in)correct behaviour of the network is needed. The error for a specific pair of input and associated target state is half the square of the length of the difference vector of target and output.

$$E = \frac{1}{2}\sum_i (target_i - output_i)^2 \qquad (1)$$

The overall measure of error is the summation of this error over all patterns; $E_{total} = \sum E$. The generalized delta rule implements a gradient descent in E. The weights are changed to reduce the error, E, by

$$\Delta W_{ji} = -\alpha \frac{\partial E}{\partial W_{ji}} \qquad (2)$$

In the remainder of this derivation it is assumed that the synapses take a weighted sum (wsum), and that the generator is a nondecreasing differentiable function, f, of the current potential. This assumption makes this derivation inapplicable to all nets with binary values. The potential of element j, v_j, is given by

$$v_j = \sum_i W_{ji}\, input_i \qquad (3)$$

The (output) value of this element is determined by

$$output_j = f(v_j) \tag{4}$$

It is useful to write the derivative in *2* as the product of the change of the error as function of the potential and the change of potential as function of the weight.

$$\frac{\partial E}{\partial W_{ji}} = \frac{\partial E}{\partial v_j}\frac{\partial v_j}{\partial W_{ji}} \tag{5}$$

From equation *3* it is obvious that the last term can be written as

$$\frac{\partial v_j}{\partial W_{ji}} = \frac{\partial}{\partial W_{ji}}\sum_k W_{jk}\, input_k = input_i \tag{6}$$

Now the function δ is defined. This function gives the learning algorithm its name.

$$\delta_j = -\frac{\partial E}{\partial v_j} \tag{7}$$

Using this definition, equation *5* can be rewritten as

$$\frac{\partial E}{\partial W_{ji}} = -\delta_j\, input_i \tag{8}$$

An equivalent form for the weight changes according to equation *2* is

$$\Delta W_{ji} = \alpha\, \delta_j\, input_i \tag{9}$$

This leaves the task to determine δ for each node in the network. Using the rule of partial derivatives again, the definition is split into a product of error change resulting from variations in the output and output changes due to variations in the potential:

$$\delta_j = -\frac{\partial E}{\partial v_j} = -\frac{\partial E}{\partial output_j}\frac{\partial output_j}{\partial v_j} \tag{10}$$

From *4* it is clear that the last factor can be rewritten as

$$\frac{\partial output_j}{\partial v_j} = f\,'(v_j) \tag{11}$$

For an output unit, the first term obtained in *10* is easily deduced from *1*

$$\frac{\partial E}{\partial output_j} = \frac{\partial}{\partial output_j}\left(\frac{1}{2}\sum_i (target_i - output_i)^2\right) = -(target_j - output_j) \tag{12}$$

By substituting these results, equation *10* becomes

$$\delta_j = (target_j - output_j)\, f\,'(v_j) \tag{13}$$

For a single layer of linear elements (the generator function is the identity function $f\,v = v$) the ordinary delta rule is obtained.

$$\Delta W_{ji} = \alpha\,(target_j - output_j)\; input_i \qquad (14)$$

The chain rule is used to derive an expression for δ of hidden units. First, the leading term in equation *10* is rewritten

$$\frac{\partial E}{\partial\, output_j} = \sum_k \frac{\partial E}{\partial v_k}\frac{\partial v_k}{\partial\, output_j} = \sum_k \frac{\partial E}{\partial v_k}\frac{\partial}{\partial\, output_j}\sum_i W_{ki}\, input_i$$

$$= \sum_k \frac{\partial E}{\partial v_k} W_{kj} = -\sum_k \delta_k\, W_{kj} \qquad (15)$$

The summation is over all units receiving output from element j. Using this result and equation *11* to rewrite *10*, δ becomes

$$\delta_j = f\,'(v_j)\sum_k \delta_k\, W_{kj} \qquad (16)$$

Using equations *13* and *16*, δ can be computed for every unit in the network. The error measure, δ, is propagated from the output units backwards. This gives rise to the name backpropagation for this algorithm. With these values, the weights can be adapted, according to *9*, to reduce the error.

When the network behaves well for all possible inputs, it will do forever; the weights are only changed when an erroneous output is generated. From *9* it can be deduced that δ must be non zero to change the weights, but from *13* and *15* it is clear that all δ's are zero when the network produces the correct output.

It is easy to show how a threshold value involved in the generation of the output can be adapted to reduce the error. Assume that the output is determined by

$$output_j = f\,(v_j + t_j) \qquad (17)$$

The error is reduced, in a gradient descent manner, by changing the threshold, t_j, by an amount

$$\Delta t_j = -\beta\,\frac{\partial E}{\partial t_j} \qquad (18)$$

Analogue to the derivation above, it can be shown that

$$\Delta t_j = \beta\;\delta_j \qquad (19)$$

Obviously, the threshold can be replaced by an additional input connected to a constant element with value β/α. Usually the value 1 is used for β/α.

Changing the weights after each presentation of an input pattern and associated target pattern departs slightly from the true gradient descent for E_{total}.

Provided that the learning rate, α, is sufficiently small, the delta rule will implement a very close approximation to the gradient descent in sum-squared error. The rule will search for the least mean square error, hence this algorithm is sometimes named the LMS-rule.

The algorithm shown here is a gradient descent method and cannot distinguish between the global minimum of the error function in the weights space and local minima. This problem resembles the problem of avoiding local maxima in any hill climbing algorithm. Fortunately, simulation results show that local minima are usually avoided.

Even when local minima are avoided, there is a problem of convergence speed. Clearly the factor α in equation 9 above, cannot be made very large to maintain the gradient descent character. A true gradient descent procedure requires even that infinitesimal steps are taken. In the discussion of networks below we will mention the learning rate and number of needed samples to teach the network some simple functions. There are also extensions proposed, to the learning algorithm as presented above, which should increase the learning speed; e.g., Rumelhart [Rumelhart 86] proposes the introduction of a momentum term in the change of weights. Recent theoretical work suggests that better results will be obtained by adapting the weights after each presentation of a pattern by a relatively large amount [Ellacott 90].

The learning algorithm presented here has another problem. When all initial weights are chosen identical in a symmetrical network and the problem desires unequal weights, the procedure will never find these weights. The error propagated back through the network is proportional to the weights: if all weights are equal all δ's will be equal. Due to the symmetry, also the outputs of the elements are equal, these outputs are used as inputs by the next layer. Equation 9 yields identical changes of the weights for all units and the symmetry is preserved forever. This problem can easily be avoided by choosing some small pseudo random values for the initial weights, or adding a little noise somewhere in the system or the learning algorithm.

The final problem is that the derivation is only valid for differentiable and nondecreasing generator functions; it not applicable in binary networks. This learning algorithm is specified in Miranda when it is used in the multi-layer perceptron described below.

In binary networks a similar learning algorithm is used. The derivation terms are simply dropped. It cannot be proven to work, but it is used with good results in many applications. This algorithm is described in detail in the discussion of multi-layer perceptron for binary values below.

Description of specific artificial neural networks

With the description tools defined above and some additional, model specific, functions it is possible to give an elegant, complete and executable description of artificial neural networks. For each network at least the following functions must be defined:

```
high                :: value                || The upper limit of the values
low                 :: value                || The lower limit of the values
nr_constant_neurons :: num                  || The number of constant neurons the net
compute_state       :: weights_net -> input_state -> [state]
compute_output      :: weights_net -> input_state -> output_state
adapt_weights       :: weights_net -> input_state -> target_state -> weights_net
gain                :: num                  || Determines the learning rate
a                   :: num                  || The scattering of the initial weights
initial_net         :: [num] -> weights_net || Generates the weights of the initial net
```

6.2 The Adaptive Linear Combiner

The ALC (*Adaptive Linear Combiner*) is the basic building stone for many adaptive systems. In terms of neural networks, the ALC is only a synapse that takes a weighted sum. It is developed independently in adaptive signal processing [Widrow 85].

```
alc :: element
alc = wsum
```

With its input connected to a tapped delay line, the ALC is the most popular key component of an adaptive filter. When the ALC is used as an adaptive filter, an estimation of the wanted output must be available to compute an error signal. This error is the difference between the system output and the desired output. The adaptive filter is tuned to reduce the error signal. There are several algorithms to decrease the error signal, the most popular method is the delta rule.

The use of these adaptive filters is widespread nowadays; all high speed modems use adaptive equalization filters, many long distance communication links are equipped with adaptive echo cancelers, furthermore these filters are used in noise cancelers and system modeling.

Since our prime interest is the description of neural networks, the ALC is not discussed in detail. It is mentioned to show that even very simple elements can perform useful tasks.

6.3 The Adaline

The name *adaline* is derived from adaptive learning element [Widrow 62]. Adalines are adaptive linear combiners cascaded with the sgn function to get a binary output [Widrow 88].

```
adaline :: element
adaline weights = sgn.wsum weights
```

An adaline computes the inner product between its weight vector and the input vector and passes this value, the potential, to the sgn function to obtain a binary output. Usually, a constant value is added to the input state to obtain an adjustable threshold.

The output of an adaline changes when the inner product crosses zero. The inner product of the input and the weight vector is zero in a plane through the origin perpendicular to the weight vector. So, the adaline checks whether the input is above or below a hyperplane in the input space. The regions in the input domain correspond to output values. Binary functions with an input domain that can be separated in two regions, separated by a hyperplane, can be performed by the adaline. This implies that an adaline can learn to execute on logical AND or OR-function, but the XOR-function cannot be performed. The typical adaline 'network' contains exactly one element.

```
compute_state :: weights_net -> input_state -> net_state
compute_state = feed_forward adaline

compute_output :: weights_net -> input_state -> output_state
compute_output weights inputs = select_output (compute_state weights inputs)

initial_net :: [num] -> weights_net
initial_net [dimension,elements]
 = [[[a*(1-((2*i+3*j) mod 7)/3)                   || Pseudo random weights
    | i <- [1..dimension+nr_constant_neurons]]  || Length of input vector
    | j <- [1  elements]]]                      || The number of elements

high = 1                                        || The standard convention  Moreover,
low  = -1                                       || it is assumed that the mean input is 0.
nr_constant_neurons = 1
```

The weight vectors for some familiar boolean functions are (the last element represents the threshold value)

```
and_vector   = [ 1, 1, - 1 5]                   || 2-input AND
nand_vector  = [-1,-1,  1 5]                    || 2-input NAND
or_vector    = [ 1, 1,  1.5]                    || 2-input OR
maj_vector   = [ 1, 1, 1, 0 ]                   || 3-input majority
```

Our specification shows that an adaline is just a special case of the perceptron. So, learning in adalines is equivalent to learning in perceptrons and it is described there.

In the early 1960s an extension to implement non linear separable functions is proposed [Ridgway 62], called *madaline* (many adalines). These networks consist of a layer of adalines connected to a single, fixed, logic device in the second layer. It can be modelled by feeding the output from the network described above to the proper function. Some useful functions are

```
and_fun :: input_state -> output_state
and_fun inputs
 = [high], if # (filter (=low) inputs) = 0
 = [low] , otherwise

majority_fun :: input_state -> output_state
majority_fun inputs
 = [high], if # (filter (=high) inputs) > # inputs/2
 = [low] , otherwise
```

A possible madaline performing the 2-input XOR-function is

```
xor_madaline = and_fun.compute_output [[or_vector, nand_vector]]
```

6.4 The Perceptron

Perceptrons [Rosenblatt 62, Minsky 69] are feed forward networks composed of simple processing elements. These processing elements are similar to the adaline described above. In the original setting, perceptrons are processing binary values. Later on a variant to process continuous variables is defined in order to apply the generalized delta rule as learning algorithm. First, the single layer binary perceptron will be described, afterwards multi-layer perceptrons for continuous and binary values will be discussed. It makes sense to distinguish single-layer and multi-layer perceptrons; the type of mappings which can be executed by these networks and the applicable learning rules are different.

The internal potential in each element of a binary perceptron is determined by the function wsum; the inner product between the input and weight vector. The generator is the hard_limit function. Usually, the threshold in this function is taken to be zero. An adaptable threshold is modelled by an additional constant input. When the bounds high and low are set to 1 and -1, which is the common choice, the single layer perceptron consists of adalines.

The single layer perceptron

In the literature, the single layer perceptron is sometimes drawn as a three layer network; a layer of input neurons yielding the input state, a layer of unalterable elements performing the fixed mapping from the input to the adaptable elements and the layer of adaptable elements. The description given here covers only the last layer, the only layer that is adapted.

The single layer perceptron contains one neuron for each element in the output state. The functional description of the single layer perceptron is identical to the adaline as described above, only the element used in compute_state is changed from adaline to perceptron_b. Binary functions with an input domain that can be separated in two regions corresponding to the output values (like the AND and OR-function) can be implemented by a single-layer perceptron.

There are effective ways to teach a perceptron to execute a function based on a set of sample inputs and the desired response. Since the generator function is not differentiable the derivation of the learning rule found as generalised delta rule is not valid. However, the normal delta rule is used to adapt the weights in this network after every presentation of a sample. This makes sense, since the generator function can be thought to magnify the error, the delta rule will improve the potentials in the network and in the long run also the output values. The correctness of this learning rule is not proven, but it appears to behave well in simulations. A set of small randomly chosen weights is chosen for the initial network. After each presentation of a sample the weights are adapted such that the error in the response of the network is decreased. The Miranda specification of the learning algorithm for the single layer binary perceptron reads:

```
adapt_weights:: weights_net -> input_state -> target_state -> weights_net
adapt_weights weights input target
 = [[[new_weight j i
    | i <- [0..# weights!l!j -1] ]    || inputs for element i in the layer
    | j <- [0..# weights!l -1] ] ]    || elements in the layer
   where  new_weight j i   =  weights!0!j!i + gain * state!0!i * error!j
          state            =  compute_state weights input
          output           =  select_output state
          error            =  [ target!i - output!i | i <- [0..# target -1]]
```

Functions with an input domain that cannot be separated by a single hyperplane (like the XOR-function), cannot be implemented by a single layer perceptron. More complicated functions must be implemented by identifying smaller regions in input space. There are at least two possibilities to implement these functions with neural networks. First, the domains in the input space corresponding to each output value can be limited by several hyperplanes, each plane can be implemented by an adaline. The information yielded by these elements can be further processed in other layers of these elements. It can be proven that these multi-layer perceptrons can learn every projection function. The second way to implement these functions is to use neurons sensitive to smaller regions in the input space. The Learning Vector Quantization network uses a set of vectors, every input vector is associated with the vector which resembles this input most.

Applications of the single layer perceptron

This single layer perceptron network learns simple two input functions like AND and OR within some tens of samples, the exact number of samples needed depends on the initial weights, the gain and the sequence of examples.

These functions are also learned when some don't-care elements are added to the input state. The weights associated with these don't-care elements become zero (or very small) as expected. Using gain = 0.1 and a scattering of initial weights a = 0.1, this net learns the two input AND-function from 20 randomly chosen input patterns. The resulting weight vector of a sample run is [0 27, 0.10,-0 23]. The last element in this vector is the connection to the constant input, representing the threshold. A three input AND-function takes 80 samples, the resulting vector is [0 27, 0 30, 0 37,-0.77]. When a don't-care term is added (the target for the function trace in appandix C is given by the function and_fun take 2) the resulting vector is [0.47, 0 30, -0.03, -0.37]. This requires 30 learning samples. The weight vectors obtained are dependent of the initial weights and the sequence of samples.

The multi-layer perceptron for continuous values

For the multi-layer perceptron, with a differentiable generator function, the generalized delta rule is used as learning algorithm. The generator function is the sigmoid defined above. This network can be specified in Miranda as:

```
compute_output :: weights_net -> input_state -> output_state
compute_output weights input = select_output (compute_state weights input)

compute_state :: weights_net -> input_state -> net_state
compute_state = feed_forward perceptron_c

adapt_weights:: weights_net -> input_state -> target_state ->weights_net
adapt_weights weights input target
 = [[[new_weight l j i
    | i <- [0..# weights!l!j -1] ]                    || inputs for neuron i in layer l
    | j <- [0..# weights!l -1] ]                      || neurons in layer l
    | l <- [0  # weights -1] ]                        || layers
   where  new_weight l j i = weights!l!j!i + gain * state!l!i * delta!l!j
          state    =  compute_state weights input
          state'
                   =  [[ sigmoid' (wsum (weights!l!j) (state!l))  | j <- [0. #weights!l -1]]
                                                                  | l <- [0  #weights -1]]
          output   =  select_output state
          output'  =  ast state'
          error    =  [output'!j*(target!j - output!j)| j <- [0  # target -1]]
          delta
                   =  [[ state'!l!j * sum [weights!(l+1)!k!j * delta!(l+1)!k
                          | k <- [0..# weights!(l+1) -1 ] ]   || outputs of neuron
                          | j <- [0  # weights!l -1] ]        || neurons in layer l
                          | l <- [0. # weights -2] ] ++       || hidden layers
                      [error]                                 || output layer
```

```
initial_net :: [num] -> weights_net
initial_net (n:m:sizes)
 = [layer]                          , if sizes = [ ]
 = layer : initial_net (m:sizes)    , otherwise
   where layer
             = [[a*(1-((613* # sizes + 337*i + 733*j) mod 101)/50)     || pseudo random
               |j <- [1..n+nr_constant_neurons]] | i <- [1..m] ]

high  =  1
low   =  -1
nr_constant_neurons =  1
```

Application of the multi-layer perceptron for continuous values

The behaviour of this network is studied, it was trained to execute some boolean functions. Using boolean functions there is a problem in deciding when the network has learned the function. Due to the used generator function the extremes high and low are never reached as output values. A large sharpness improves the separation between the output values, but tends to slow down the recovery from wrong situations since the derivative tends to zero very soon. We have used as parameters a=0.2, gain=0.5 and sharpness=2. This results in a quick convergence for the three input majority function, within 100 samples the output values are within 10% of the difference between high and low of the correct values. The initial vector with value [-0.04,-0.14,0.16,0.05] is changed to [1.47,1.46,1.45,0.14]. A two layer network with two hidden units and one output unit learns the XOR-function within 190 samples. The initial weights generated by initial_net [2,2,1] are [[[0.08,-0.02,-0.02], [-0.05,-0.16,0.14]], [[-0.04,-0.14,0.16]]]. The obtained weights are [[[1.21,-1.25,-0.90], [1.85,-1.56,1.85]], [[1.71,-1,67,1.47]]]. To show how these vectors implement the XOR-function the truth tables for the processing elements are shown; the output values are rounded to boolean values. Elements 1 and 2 are hidden, element 3 is the output neuron.

inputs		element 1	element 2	element 3	system	target
low	low	low	high	high	low	low
low	high	low	low	low	high	high
high	low	high	high	high	high	high
high	high	low	high	high	low	low

Multi-layer perceptrons with the backpropagation learning rule are used with success in many practical applications [see for instance INNC 90]. The main problem is to develop a suitable mapping from the problem variables to the neural network inputs and to use the network output in the appropriate way. The number of hidden units needed, the gain and the initial network are determined empirically.

The multi-layer perceptron for binary values

One way to ensure that the output reaches the bounds high and low is to use a binary generator function. Again, there is no learning rule which can be proven to reduce the error, but a generalisation of the single layer learning rule for binary values, the delta rule, can be used here. Compared with the generalized delta rule, this comes down to dropping the derivation terms. The single layer perceptron is just a special case of this network. This network differs from the network for continuous values in the element used to compute the state and the learning rule:

```
compute_state :: weights_net -> input_state -> net_state
compute_state = feed_forward perceptron_b

adapt_weights :: weights_net -> input_state -> target_state ->weights_net
adapt_weights weights input target
 = [[[new_weight l j i
    | i <- [0..# weights!!l!j -1] ]                    || inputs for neuron j in layer l
    | j <- [0..# weights!!l -1] ]                      || neurons in layer l
    | l <- [0..# weights -1] ]                         || layers
   where  new_weight l j i = weights!!l!j!i + gain * state!!l!i * delta!!l!j
          state   =  compute_state weights input
          output  =  select_output state
          error   =  [target!j - output!j | j <- [0..# target -1]]
          delta
                  =  [[sum [weights!(l+1)!k!j * delta!(l+1)!k
                        | k <- [0.. # weights!(l+1) -1 ] ]  || outputs of neuron
                        | j <- [0..# weights!!l -1] ]       || neurons in layer l
                        | l <- [0..# weights -2] ] ++       || hidden layers
                     [error]                                || output layer: error
```

A three layer perceptron can perform every projection of the input state to the output state provided there are enough elements in the layers [Minsky & Papert 69]. The first two layers select the proper regions in the input space while the third layer combines the output of these combinations. Unfortunately, the amount of processing elements required cannot be determined.

There is a very crude implementation of *every* boolean function on a two-layer perceptron [Coolen 89]. Its use is not recommended, but it shows the power of multi-layer perceptrons. The implementation of the function for each output bit is done with brute force:

- For every possible input pattern that corresponds to a high output value there is a hidden neuron that selectively responds to that pattern. This is done by making the weight vector identical to the input pattern and choosing the proper threshold (square of the length).

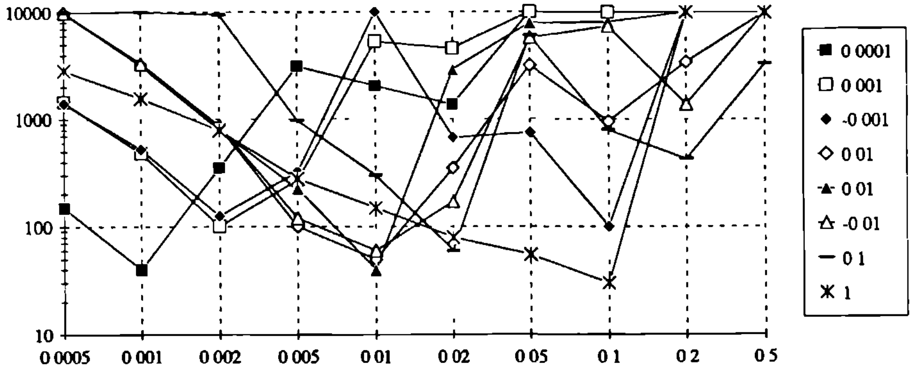
- The output bit can now be determined by a perceptron taking the logical OR of these hidden units.

Application of the multi-layer perceptron for binary values

Using this network description, the number of training samples needed to learn the XOR-function as function of the amplitude of initial weights, a, the learning rate, gain, and the number of neurons in the hidden layer is explored. The XOR-function is used because it is a very simple and small function, but still interesting through a non linear separable input-output releation. Experiments were done with the same pseudo random sequence of samples and the same set of pseudo random initial weights (except for the amplitude). To study the influence of the particular choice of initial weights some additional experiments were done; a negative amplitude inverts all initial weights and a different set of initial weights is used as additional measurement, these results are printed in italics. After several learning samples the performance of the system was evaluated until the XOR-function was learned. The number of samples between the subsequent evaluations of the network performance is always less then 10 % of the total number of samples needed. The Miranda system on a SUN-3 performs around 200 updates of individual connection strength per second, including evaluation of the performance and printing statistics. Our patience and the amount of heap space used limit the number of learning cycles somewhere around 10000. The number of samples corresponding to the optimal gain for a given amplitude is printed bold.

↓a|gain→	0.0005	0.001	0.002	0.005	0.01	0.02	0.05	0.1	0.2	0.5
0.0001	150	**40**	350	3100	2050	1350	>10000	>10000	>10000	
0.001	1450	480	**100**	270	5270	4575	>10000	>10000	>10000	
-0.001	1400	525	125	325	>10000	675	750	**100**	>10000	
0.01	9850	3200	850	100	**50**	350	3175	950	3400	>10000
0.01	*>10000*	*3175*	*800*	*225*	***40***	*2875*	*7875*	*8100*	*>10000*	*>10000*
-0.01	>10000	3300	875	120	**60**	170	5750	7350	1375	>10000
0.1	>10000	>10000	9525	975	300	**60**	6000	800	425	3300
1	2850	1550	790	280	150	80	55	**30**	>10000	>10000

Number of teaching samples needed by a 2-layer binary perceptron to learn the 2-input XOR-function with 2 elements in the hidden layer.



The data from the table above shown graphically. Vertical the number of samples needed, horizontal the gain.

↓a|gain→	0.0005	0.001	0.002	0.005	0.01	0.02	0.05	0.1	0.2	0.5
0.0001	**95**	925	800	5625	175	3800	2500	>10000		
0.001	1475	310	**70**	150	550	350	6650	>10000		
0.01	>10000	3675	1900	80	**45**	1800	6625	650	6950	>10000
0.1	>10000	>10000	>10000	1825	450	210	**40**	3200	1900	1500

Number of teaching samples needed by a 2-layer binary perceptron to learn the 2-input XOR-function with 3 elements in the hidden layer.

↓a|gain→	0.0005	0.001	0.002	0.005	0.01	0.02	0.05	0.1	0.2	0.5
0.0001	150	900	**80**	2675	1250	400	850	750	>10000	
0.001	875	510	**125**	1240	5600	1250	850	750	>10000	
0.01	9800	287	525	100	**45**	3775	5350	1725	4850	8725
0.1	>10000	>10000	5025	1125	375	**160**	225	300	3400	3800

Number of teaching samples needed by a 2-layer binary perceptron to learn the 2-input XOR-function with 4 elements in the hidden layer.

↓a|gain→	0.0005	0.001	0.002	0.005	0.01	0.02	0.05	0.1	0.2	0.5
0.001	>9400	>9400	>9400	>9400	>9400	4150	3700	4750	**1700**	>10000
0.01	>9400	>9400	>9400	>9400	6375	>9425	2100	1800	3100	**1700**

Number of teaching samples needed by a 2-layer binary perceptron to learn the 3 input XOR-function with 2 elements in the hidden layer.

↓a|gain→	0.0005	0.001	0.002	0.005	0.01	0.02	0.05	0.1	0.2	0.5
0.001	1225	300	**150**	800	3250	6100	7450	>10000	700	>10000
0.01	>9175	3100	800	>9125	**85**	>9050	3600	700	600	1600

Number of teaching samples needed by a 2-layer binary perceptron to learn the 3 input XOR-function with 3 elements in the hidden layer.

From these figures some tendencies can be observed:

- The 2-input XOR-function is quickly learned for each number of hidden units tried, provided that the initial parameters are well chosen.
- Amplitude and gain can be chosen in a wide range of values to achieve a correctly behaving network. There is however, a clear optimum gain for each amplitude. This optimum is largely independent from the actual initial weights; the amplitude of these weights is much more important. The optimum gain increases when the amplitude of the initial weights increases.
- The 3-input XOR-function appears to be much more difficult than the 2-input XOR-function. The 3-input network behaves nearly correct (only one pattern gives the wrong answer) within some hundreds of samples, but it requires much time to become totally correct. The behaviour of the network keeps changing in time, so it is not stuck in a local minimum of the error function.

The internal representations of the XOR-function can be deduced from the obtained weights. There is not a unique internal representation for a given network, but the representation changes among the learn runs.

6.5 Learning Vector Quantization

The Learning Vector Quantization (LVQ) network [Kohonen 88] consists of a single layer of elements. All elements determine the correspondence between the input vector and their weight vector by computing the inner product. The output of the network is the class associated with the element with the largest potential. When several elements produce an equal potential a choice is made; here the element with the lowest index number is used. The element with the largest potential can be determined in the neural network manner by a competition of elements in an N-flop feedback network (see below), or in a more direct way. Here the direct way is used; the maximum potential is determined and the winner is the first element with that potential.

During the learning phase the winning weight vector is adapted; when the output class is equal to the target the weight vector is changed in the direction of the input to increase the inner product further. In case of an erroneous output, the winning weight vector is changed in the opposite direction to decrease the potential of the winning element.

There is no need to use random vectors as initial values for the weights in this network. It appears that the best results are obtained when all initial vectors are chosen in the middle of the input space.

Unlike learning according to the delta rule, this network is adapted when it behaves correctly for the current input-target pair. As a consequence, it is possible that the network performance decreases even if it had learned to execute its task perfectly. This can be understood by considering a vector responsible for the correct mapping of inputs in some region in the input space, a sequence of inputs on one side of this region drifts the weight vector to this side. The other side of the region becomes closer to another weight vector and hence, possible to another output class. This happens only when a single weight vector maps several different inputs to their targets. This effect can be reduced by slowly decreasing the learning rate.

```
vector            ==  [num]
vector_id         ==  num
class             ==  state
target_class      ==  class
nr_off_classes    ==  num

associate_class :: vector_id -> class
associate_class id = [id mod nr_off _classes]      || equal distribution over classes

compute_state :: [vectors] -> input_state -> net_state
compute_state = feed_forward inner_product

compute_output :: [vectors] -> input_state -> output_state
compute_output vectors input = associate_class (find_winner vectors input)

find_winner :: [vectors] -> input_state -> vector_id
find_winner vectors input = id_max (select_output (compute_state vectors input))
                  || more direct: id_max [inner_product vector input | vector <- vectors]

id_max :: output_state -> vector_id
id_max state = scan_list 0 (max state) state

adapt_weights :: [vectors] -> input_state -> target_class -> [vectors]
adapt_weights [vectors] input target
 = [[new_vector i | i <- [0..# vectors -1]]]
   where
   new_vector i
      =  vectors!i                                , if i ~= winner
      =  [vectors!i!j + gain * input!j | j <- elements]   , if associate_class winner = target
      =  [vectors!i!j - gain * input!j | j <- elements ]  , otherwise
   elements   =  [0..# vectors!i -1]
   winner     =  find_winner [vectors] input

initial_net :: [num] -> [vectors]
initial_net [input_dimension,size]
 = [[vector i | i <- [1..size]]]
   where   vector i
                  =  [(high+low)/2 + a*((2*j+3*i) mod 7)/6
                     | j <- [1..input_dimension + nr_constant_neurons] ]

high  =  1                          || the usual assumption
low   =  -1
nr_off_classes = 2                  || binary output
```

```
nr_constant_neurons = 0
```

For vectors not distributed at a fixed distance around the origin, the inner product is not applicable as a measure of correspondence. The length of the difference vector can be used as a measure of correspondence in this situation [Kohonen 88]. The element with the least potential must be the winner in this situation. In the learning algorithm described above the weight vectors can grow unbounded to resolve some severe errors of the network. The learning algorithm has been adapted to decrease the length of the difference vector when the network produces the correct answer.

```
compute_state :: [vectors] -> input_state -> net_state
compute_state = feed_forward difference

difference :: vector -> vector -> num
difference v1 v2 = length [v1!i - v2!i | i <- [0  #v1-1]]

compute_output :: [vectors] -> input_state -> output_state
compute_output vectors input = associate_class (find_winner vectors input)

find_winner :. [vectors] -> input_state -> vector_id
find_winner vectors input = id_min (select_output (compute_state vectors input))

id_min ·· output_state -> vector_id
id_min state = scan_list 0 (min state) state

adapt_weights :: [vectors] -> input_state -> target_class -> [vectors]
adapt_weights [vectors] input target
 = [[new_vector i | i <- [0. # vectors -1]]]
   where
   new_vector i
    = vectors!i                                        , if i ~= winner
    = [(1-gain)*vectors!i!j+gain*input!j | j <- elements ], if associate_class winner = target
    = [vectors!i!j - gain*input!j | j <- elements ]          , otherwise
   elements   =  [0  # vectors!i -1]
   winner     =  find_winner [vectors] input
```

There are many similar networks proposed in the literature. The Hamming network [Lippmann 87] uses the Hamming distance between the input bit vector and stored vector to select the best matching vector. The Hamming distance between two bit vectors is the number of corresponding elements that are unequal. The counter propagation network [Hecht-Nielsen 87a] does not use a fixed association of classes to vectors, but this relation is learned in a perceptron like manner from the samples.

Applications of Learning Vector Quantization

The LVQ network learns very quickly. Using the network description given above the number of teaching samples needed for various values of the initial values and gain is determined. The 2-input XOR-function was taught to a network consisting of 4 vectors. Several variants of the learning algorithm are compared. In the first table the original learning algorithm is used. The second table contains the results obtained using the second algorithm. In the last learning algorithm the vectors corresponding to the right classes are normalised. The performance of the last algorithm decreases very significantly if the wrong vectors are normalised after adaptation. The network becomes extremely sensitive for the choice of the initial values and the sequence of training inputs. The Miranda system on a SUN-3 performs around 30 vector updates per second.

↓a|gain→	0.0001	0.001	0.01	0.1	1	0.0001	0.001	0.01	0.1	1
0	21	21	21	21	21	19	19	19	19	136
0.0001	18	18	18	11	11	11	11	11	11	3
-0.0001	6	19	19	18	5	4	5	5	4	5
0.001	18	11	18	11	18	11	11	12	11	12
-0.001	4	4	4	18	4	4	4	4	2	2
0.01	18	3	11	11	18	12	3	11	11	12
-0.01	6	4	4	4	4	5	4	4	2	2
0.1	68	18	3	11	18	68	12	3	11	12
-0.1	86	6	4	4	19	81	5	4	4	5
1	800	68	18	7	18	550	58	11	4	11
-1	830	82	6	6	6	570	51	5	4	4

Left: Number of teaching inputs required by a 4-vector learning vector quantization network to learn the 2-input XOR-function. The original learning rule is used.

Right: Initial values and trainings set as above. The second learning rule is used for this LVQ network, this rule yields vectors of bounded size.

↓ initial vectors | gain →	0.0001	0.001	0.01	0.1	0.5	0.9999
all vectors [1,1]			>8575	>10000	22	31
all vectors [0,1]	5	5	5	5	5	5
pseudo random vectors	>6500	2	2	2	11	11
inverse of vectors above		>10000	>10000	6	4	11

The same samples are used again to teach this LVQ network. Now the vectors are normalised after a successful recognition in order to keep their length finite.

From these figures it is clear that the learning vector network learns the XOR-function much faster than a multi-layer perceptron using the backpropagation learning rule. Only when the scattering of initial vectors is much larger than the gain a large number of learning cycles is needed. The second learning rule keeps the length of the vectors finite and performs slightly better than the others so it is recommended to use that one. It behaves much better when the distribution of samples over the vectors is not uniform. The normalising variant appears to be sensitive for the initial values and is not applicable when one of the vectors becomes the 0-vector. This variant does behave well, when the bounds low and high are set to 0.1 and 1.0.

6.6 Feedback Networks

Networks with an *internal* loop structure, a connection of (part of) the outputs of some elements back to their inputs, are called feedback networks. All networks can be applied in a feedback configuration by making an *external* connection between the output and the input.

Feedback networks became famous from the work of Hopfield [Hopfield 82], although they were first used by Kohonen and Little [Kohonen 77, Little 74]. These networks consist of a feedback configuration of McCulloch-Pitts neurons. These elements take a weighted sum of their input values and yield usually a binary output value. When the weight vectors of all neurons are viewed as a matrix, the potentials are computed by multiplying this matrix with the input vector. The output vector is obtained by mapping the generator function over the potential vector. The state is cycled through the network until a stable state is reached.

The Hebb [Hebb 49] learning rule is used to tune this network. The strength of a connection is increased when the connected input and target value correspond, otherwise the weight is decreased. The connections from a neuron to its own input are kept zero in the historical setting, although the network behaves well with a non zero diagonal term in the weight matrix. Using this learning rule the network becomes an associative memory. The network can be used for pattern recognition. The proper weights are usually obtained by using the Hebb rule with each pattern used simultaneously as input and target. A pattern is also learned from multiple presentations, each one polluted with random noise. The patterns become the eigen vectors of the matrix multiplication cascaded with the generator function. The network seeks the pattern corresponding to an input state by re-applying the function until a fixed point is reached. The maximum number of patterns that can be stored in a large weight matrix is 14% of the number of neurons, provided that the patterns are not too similar [Hopfield 82, Amit 85].

Simulations show that the network finds the patterns faster when the neurons are updated one by one. Each neuron uses the most recent state to determine its output and unstable equilibrium points are avoided by the pseudo random evaluation order.

The bounds low and high are chosen nowadays usually -1 and 1, but Hopfield used 0 as lower bound. This symmetry has as result that the inverse of a learned pattern is also a fixed point of the system.

```
compute_state :: weights_net -> input_state -> net_state
compute_state = feed_back_asyn hopfield_b

compute_output :: weights_net -> input_state -> output_state
compute_output weights input = select_output (compute_state weights input)

adapt_weights:: weights_net -> input_state -> target_state ->weights_net
adapt_weights weights_net input_state target_state
 = [[[new_weight i j | j <- [0..n-1] ]  | i <- [0..n-1] ] ]
   where  new_weight i j
                = 0                                                    , if i=j
                = weights_net!0!i!j + gain * target_state!i * input_state!j  , otherwise
          n  =  # (weights_net!0)

initial_net :: [num] -> weights_net
initial_net [n]
 = [[[a*(1-((2*i+3*j) mod 7) / 3) | i <- [1..m] ] | j <- [1..m] ] ]      || pseudo random
   where m = n + nr_constant_neurons                                     || square matrix

high  =  1
low   =  -1
nr_constant_neurons = 0
```

The system appears to be sensitive for transformations of the input state. A translated pattern is not recognized. This can be partly solved by learning the network the inverse of the transformation expected. A large number of random patterns is used as sample while the target is the transformed pattern. It is of course, possible and more efficient to start with a matrix that performs the desired transformation. For transformations it is clearly better to use a synchronous update of the neurons, here an ordinary matrix multiplication is desired. When transformation and pattern recognition are taught in the right ratio, the network transforms an input pattern until it is recognized.

Convergence

In order to show the convergence of the state in asynchronously updated binary feedback networks, an energy is associated with each state s of the system. It will be shown that stable states are energy minima and that the system will evolve to a minimum. The energy E is defined by

$$E = -\frac{1}{2}\sum_{ij} s_i\, w_{ij}\, s_j \qquad (1)$$

The output for element i is given by

$$s'_i = sgn\,(v_i) \qquad (2)$$

$$v_i = \sum_j w_{ij}\, s_j \qquad (3)$$

It will be shown that the energy decreases when the state is changed according to the given rules, until a minimum is reached. It is required that the weight matrix is symmetric, which is achieved automatically by the Hebb learning rule for pattern recognition provided that the bounds -1 and 1 are used and the initial matrix is symmetric. The change in energy due to the evaluation of element k is (all other elements remain unchanged)

$$\begin{aligned}
\Delta E &= E' - E && \{\textit{using equation 1}\}\\
&= -\frac{1}{2}\sum_{ij} s'_i\, w_{ij}\, s'_j + \frac{1}{2}\sum_{ij} s_i\, w_{ij}\, s_j && \{\textit{drop identical terms}\}\\
&= -\frac{1}{2}\Big(\sum_i s'_i\, w_{ik}\, s'_k + \sum_j s'_k\, w_{kj}\, s'_j + \sum_i s_i\, w_{ik}\, s_k + \sum_j s_k\, w_{kj}\, s_j\Big) &&\\
& && \{w_{ii} = 0\}\\
&= -\frac{1}{2}\Big(\sum_i s_i\, w_{ik}\, s'_k + \sum_j s'_k\, w_{kj}\, s_j + \sum_i s_i\, w_{ik}\, s_k + \sum_j s_k\, w_{kj}\, s_j\Big) &&\\
& && \{\textit{collect terms}\}\\
&= -\frac{1}{2}\Big(\sum_i s_i\, w_{ik}\, (s'_k - s_k) + \sum_j (s'_k - s_k)\, w_{kj}\, s_j\Big) && \{\textit{rearranging}\}\\
&= -\frac{1}{2}(s'_k - s_k) \sum_j (w_{kj} + w_{jk})\, s_j && \{w_{ij} = w_{ji}\}\\
&= \quad (s'_k \;\; s_k) \sum_j w_{kj}\, s_j && \{\textit{using equation 3}\}\\
&= -\, (s'_k - s_k)\, v_k && (4)
\end{aligned}$$

It is easy to see that $\Delta E = 0$ when $s'_k = s_k$. If the output of element k is changed, than $s'_k = -s_k$ and using equation 2 one finds $\Delta E = -2\, s'_k\, v_k < 0$. Since E is bounded, the system will evolve to a (local) minimum of E.

The observation that a Hopfield network searches a minimum energy in the system space leads to several new applications. The network is now used to find the best solution for some assignment problem instead of pattern recognition [Hopfield 85]. Well-known examples are the travelling salesman problem and job assignment for people with different rates for the tasks. An energy surface is defined that corresponds to the problem. An N × N-network is used for a problem of size N. First, N N-flops are built. An N-flop is the N dimensional analogue of a flip-flop, it is a collection of N neurons which are only in a stable state when one of them is active. This is achieved by inhibitory connections between the neurons. These N-flops are used to achieve that each site is visited only once in the TS-problem or each task is assigned to exactly one person in the job assignment problem. The N-flops are interconnected with a strength depending on the problem; the distance between the cities or the ranking of people for the tasks. When the network is in a stable state the active neurons show the solution found by the network; the best path for the salesman or the optimum task assignment. Unfortunately, this optimum found can correspond to a local energy minimum, then the assignment is a sub-optimal one. In some practical situations such a solution may be acceptable. Networks using continuous valued generator functions appear to avoid local minima better than the corresponding networks with binary generator functions. The derivation of the convergence shown above is not applicable for networks with continuous values.

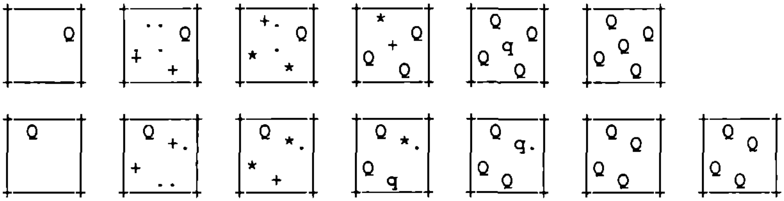
Application of a feedback network

We will illustrate the use of Hopfield networks by constructing a feedback net that searches a solution of the N-queens problem; place N queens on an N × N-chess board, such that they cannot hit each other. Of course, an N × N network is used for this purpose. The connections are arranged such that the queens are repulsive; the potential of a field is decreased when it is covered by a queen on another field. Furthermore, there is a tendency in the generator function to place queens on the board when a field is not covered by any other queen. In order to try to avoid local minima, we use pseudo linear outputs and a diagonal term to decrease the changes between the various steps. In this way a hill climbing system is built. The asynchronous evolution function is used to compute each new state.

```
queen_generator :: generator
queen_generator v = clip (v + 1/3)

queen_element :: element
queen_element weights = queen_generator.wsum weights

compute_state :: weights_net -> input_state -> net_state
compute_state = feed_back_asyn queen_element

compute_output :: weights_net -> input_state -> output_state
compute_output weights input = select_output (compute_state weights input)

initial_net :: [num] -> weights_net
initial_net [n] = [[[matrix_value i j n | i <- [0..(n^2-1)]] | j <- [0..(n^2-1)]]]

matrix_value position1 position2 dimension
 = 1      , position1 = position2                  || diagonal term
 = -1/2   , i=k V j=l V                            || rook-like attack
            abs (i-k) = abs (j-l)                  || bishop-like attack
 = 0      , otherwise                              || no attack
   where  (i,j)  =  coordinates position1 dimension
          (k,l)  =  coordinates position2 dimension
```

```
coordinates field_number board_size
 = (row,col)
   where  row  = field_number div board_size
          col  = field_number mod board_size

high  = 1.0
low   = 0.0
nr_constant_neurons = 0
```

The network described can be used to change an initial configuration to a solution with the functions defined in appendix C. To show the behaviour of this network, the results of two evaluations are reproduced here. Note that the initial configurations are isomorphic upon a rotation over 90°. The increasing probability to place a queen on a field is indicated by the characters ' ', '.', '+', '*', 'q' and 'Q'.



Unfortunately, the system gets frequently stuck in a local minimum: a stable state with too few queens on the board. The standard solution to this familiar problem is to add a noise term which enables the system to leave the local minima. This term is decreased each iteration and one hopes that the system cools down in the absolute energy minimum. The terminology is borrowed from physics due to the similarity with thermal noise in spin models.

Avoiding local minima

The Hopfield network with binary values is isomorphic [Amit 85] to the Ising spin model for many coupled spin particles [Reif 65, Reichl 80]. There is a famous technique in physics to find the minimum energy state for such a system called simulated annealing. The system starts at a high temperature which implies that much noise is added when a new system state is computed; there is a large chance to change spin. While new states are computed, the temperature is slowly decreased. It can be shown that the expectation value of the system state is the minimum energy state. This technique is also used in feedback networks to leave local energy minima [Kirkpatrick 83]. Unfortunately, this does not imply that a feedback network will always find the minimum energy state when a slowly decreasing amount of noise is added during the iterations.

We will show that the addition of a decreasing noise term to the states helps the system to find solutions for the queens problem. A real simulated annealing procedure requires binary neurons and a state change proportional to exp(-ΔE/T) for uphill state changes, state changes that reduce the energy are always accepted. The original Hopfield networks are the T = 0 limit of this rule.

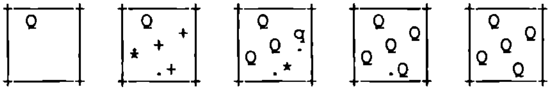
```
compute_state :: weights_net -> input_state -> net_state
compute_state = annealing 0.5 queen_element

annealing :: num -> element -> weights_net ->input_state->[output_state]
annealing t element [weights] input
 = to_limit (disturb t (eval_one_by_one element weights) input)

disturb :: num -> (state->state) -> state -> [state]
disturb t eval state
 = state : disturb (annealing_rate * t) eval new_state
   where  new_state =  eval [state!i + noise t (i+v) | i <- [0..#state-1]]
          v         =  sum state
```

For binary units it is better to flip to values in the state with a decreasing probability instead of adding the noise term.

```
noise :: num -> num -> num
noise t s = t*(1-(random (23459*entier s) mod 101)/50)   || pseudo random [-t,t]
```

Avoiding local minima in the N-queens example

The result of the evaluation of second initial state, using this new network definition with an annealing rate of 0.8, is shown below.



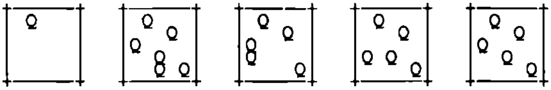
The network behaves better using this evaluation function. Although, it is most likely to find an absolute energy minimum, it cannot be assured that the system ends up in a state with N queens. We have not bothered ourselves with the optimization of the initial 'temperature' and annealing rate of this network. The current choice of parameters works for a number of cases, but certainly not always. The relatively poor performance of this network is not surprising; as indicated above local minima cannot be avoided by this network. It should only be used in situations where a sub-optimal solution is acceptable.

An effect similar to the addition of noise can be achieved by making large steps in the gradient decent algorithm. The generator function and the weights are changed compared to the first attempt.

```
queen_generator v = clip     (v+3)        || a pseudo linear or,
queen_generator v = sigmoid  (v+3)        || a differentiable generator function
```

```
matrix_value position1 position2 dimension
 = -2    , position1 = position2                     || diagonal term
 = -2    , i=k V j=l V abs (i-k) = abs (j-l)         || attack
 = 0     , otherwise                                 || no attack
   where  (i,j)  = coordinates position1 dimension
          (k,l)  = coordinates position2 dimension
```

Using the first queen_generator function listed above the evolution of the network becomes:



Using this network a correct solution is found for all start configurations isomorphic to the shown form. Andres reports very good results for these weights, but he uses the second generator function [Andres 90]. He shows a derivation for the form of the weights in the matrix from an appropriate energy function. The magnitude of these weights is set in a trial and error process. A network with a sigmoid generator function needs more iterations to come near to a good solution, but seems to avoid local minima a bit better. Wrong solutions caused by getting stuck in a local minimum appear to occur only in small networks [Andres, private communications]. The number of iterations needed appears to increase very slowly with the size of a network. The 10-queens problem requires on average 69 iterations, the mean number of iterations needed to solve the 300-queens is 205.

When this solution is compared with an ordinary backtrack solution of the 8-queens problem in a functional language, the performance of the neural network is quite disappointing. The backtrack solution finds the 92 solutions to the 8-queens problem within approximately the same time as it takes to compute the weight matrix for the neural network algorithm! The backtrack algorithm used is given below.

The search space for the backtrack algorithm is reduced by an efficient coding of occupied positions. A simple analysis shows that each row must contain exactly one queen. So, the solution can be described by a list of positions of the queens in each row. A further analysis shows that also each column must contain one queen. Hence, the solutions are permutations of the numbers [0..n-1], where bishop-like attacks are prevented. The empty list of queens is extended in all possible ways until all queens are placed.

```
size              ==  num          || the size of the board
solution          ==  [position]   || the position of the queen on each row
partial_solution  ==  [position]
position          ==  num          || queen position on a row; column number
```

```
queens :: size -> [solution]
queens n = extend [ ] [0..n-1]

extend :: partial_solution -> [position] -> [solution]
extend part unused
 = [part]                                                              , if unused = [ ]
 = concat  [ extend (q:part) (filter (~= q) unused)
           | q <- unused ; and [abs (q - part!i) ~= i+1|i <- [0..#part-1]]]    , otherwise
           || try each remaining position in next row; test for bishop-like attacks
```

6.7 Discussion

Functional languages appear to be very well suited to describe neural networks, although the descriptions presented here are longer than the customary descriptions using constructive mathematics. Mathematics is also more convenient for the proofs shown in this chapter.

The implementation effort required for these network descriptions is small and it yields a number of advantages. The obtained specifications are executable, partial correct and complete. This in contradiction to the usal mathematical descriptions where, for instance, the bounds for the summations and the evaluation order of elements is not specified. Apart from these usual advantages there are some additional benefits; due to the used framework of description tools the similarities and differences between the described networks are made explicit. These tools appear to be suitable to describe a large number of networks as shown in this chapter. Furthermore, these tools are expected to be suited for the description of many other kinds of networks. The described neural networks have indeed many similarities; all networks discussed use McCulloch-Pitts neurons arranged in a layered structure. Each network uses a fixed type of elements. The only difference among the elements in a network is the weight vector used to specify the individual elements.

The executability of the neural networks specifications presented here, is particularly convenient to determine properties which cannot be deduced analytically, but must be determined empirically. The stochastic behaviour, found in many networks, can be simulated by a pseudo random generator. To obtain really stochastic behaviour, a random generator must be added to the functional language, although this spoils the referential transparency.

In spite of the very simple processing elements used and the limited possibilities to prove that the artificial neural networks will work, these networks appear to be able to perform a number of stiff tasks. But, the large number of teaching samples needed by perceptrons and the unpredictable changes in this number are a severe problem in the application of these neural network models in computer science. A single pass through the network requires much computation; usually quadratically increasing (or worse) with the size of the input pattern. It is in principle possible to construct special purpose hardware to speed up the evaluation of a neural network.

A LVQ network has more appealing properties; the learning speed is usually much higher and more constant. Moreover, the amount of computations required is smaller. Unfortunately it requires some initial knowledge about the number of classes required and the number of patterns in these classes; the number of vectors needed in a given class cannot be predicted. Moreover, the learning algorithm used in these networks does not always work. It is even possible that a network which behaves totally correct 'forgets' what it has learned.

Since the artificial neural networks treated in this chapter can be modelled adequately in a functional programming language the set of functions that can be computed by these networks is smaller or equal to the functions expressible in a functional programming language. However, the character of programming is quite different. A neural network is trained to map similar inputs to similar outputs, a correct behaviour is only obtained by a very rich set of trainings samples. An ordinary program is a generally applicable algorithm.

Chapter 7
Summary and Conclusion

The suitability of functional programming languages as a formalism for executable specifications is investigated in this thesis.

A specification is a detailed product description in a suited formalism. Specifications are used in computer science to describe the functional and operational behaviour of algorithms and machines. A good specification is consistent, correct and sufficiently complete. A suited specification formalism is unambiguous and yields clear and compact descriptions.

A prototype is an implementation that obeys (some aspects of) the specification. It is constructed in order to achieve confidence in of the suitability of the specified system and in the correctness of the specification.

Using a programming language as description formalism has a number of advantages. Programming languages have usually a well defined and clear semantics, yielding unambiguous and understandable descriptions. The implementation of the programming language can be used to check some correctness aspects of the specification. A compiler cannot examine whether a specification is correct, but it can at least spot syntax errors, undefined identifiers and type conflicts. Last but not least, the description is executable, so a specification is its own prototype.

Despite of these benefits programming languages are seldom used as description formalism. The expressive power of most programming languages appears to be too low to obtain a concise description. Due to the lack of suited description primitives much programming effort is needed. The size of the obtained description is too large to be clearly readable. Moreover, the distinction between the specification and the product vanishes.

Functional programming languages are advertised as high level languages with a great expressive power. Hence they seem to be suited as description formalism. We have tried the use of functional languages as description formalism in practice. On one hand we have compared descriptions in functional languages with some other description formalisms, on the other hand we have applied them in some complex real-life applications.

In chapter 2 a hierarchical machine description in the functional programming language Miranda is compared with a traditional description. It appears that the functional specification is a bit longer. This is caused by the layout rules we have imposed ourselves to obtain the clearest description. The functional description is more direct than the imperative description. Moreover, it is very understandable and appears to be a useful prototype.

This description method is applied with success in chapter 3 to a more complex concrete problem: the specification of an abstract imperative graph rewrite machine. This machine and its description are used in practice as an intermediate level in the implementation of functional languages. The obtained specification is actually the one and only definition of this machine. It is found that it is easy to experiment with slightly different instruction sets in this prototype.

In chapter 4 the suitability of functional languages for the description of program translations is investigated. A program translation in Miranda is compared with some conventional description methods. The specification in a functional language is at least as clear as the other descriptions.

The translation of the graph rewrite language Clean to code for the machine defined in chapter 3 is specified with such a specification in chapter 5. This specification was a useful guide-line for the construction of the compiler. Although not all optimizations implemented are described, it explains the kind of code generated very well. This description was used together with the prototype machine to execute Clean programs before the corresponding products were made.

The artificial neural network descriptions presented in chapter 6 are implementations of the neural models. The specification is a useful addition to the customary specification in a mathematical style. This mathematical specification gives some dependencies of values, but no constructive way to combine these dependencies to a reasonable efficient algorithm. Moreover, there are often inaccuracies, like omitted bounds for a summation, in these mathematical specifications. These mathematical specifications can be easily recognized as a part of the functional description. A set of description tools is presented and applied to describe a number of networks. The similarities and differences between the various networks are easily spotted in this uniform description.

In conclusion it can be stated that functional specifications are applied successfully in a number of real applications. Comparison with conventional descriptions shows that the overhead imposed by using a general purpose functional language instead of tailor made formalism is limited. The obtained specifications are clear descriptions and usable prototypes.

Little attention is given to formal proofs of the correctness of the specifications. Due to the size of the applications it is a tedious job to derive and verify such proofs. However, functional specifications are suited for the rewrite type of proofs shown in chapter 2. The equivalency of terms can in fact be checked by the prototype implementation. The proofs in chapter 6 are given in ordinary mathematics; there is still too little experience with symbolic manipulation of functional programs to perform these proofs with the functional specification.

The general purpose functional language Miranda is used for the specifications presented in this thesis. It appears to be reasonably suited as specification formalism. There are, however, two problems with this language. Due to the lack of rewrite semantics partially parametrized functions are not treated as proper root normal forms. This hampers the presentation of the instructions in our machine descriptions. Secondly, the Milner/Mycroft type system forces us to inject some superfluous constructors in the data structures used to represent programs for the transformations in the chapters 4 and 5.

Appendix A
Definition of the ABC-Machine

This appendix contains the complete specification of the ABC-machine and ABC-assembler. Moreover some additional functions to obtain a better usable prototype implementation are included. These topics are discussed in chapter 3.

A.1 The micro-instruction set

```
abstype      astack, bstack, cstack, graphstore, descstore, pc, programstore, io
  with
```

Definition of the A-stack

```
as_get      :: a_src -> astack -> nodeid
as_init     :: astack
as_popn     :: nr_args -> astack -> astack
as_push     :: nodeid -> astack -> astack
as_pushn    :: nodeid_seq -> astack -> astack
as_topn     :: nr_args -> astack -> nodeid_seq
as_update   :: a_dst -> nodeid -> astack -> astack
```

Definition of the B-stack

```
bs_copy     :: b_src -> bstack -> bstack
bs_get      :: b_src -> bstack -> basic
bs_getB     :: b_src -> bstack -> boolean
bs_getI     :: b_src -> bstack -> int
bs_init     :: bstack
bs_popn     :: nr_args-> bstack -> bstack
bs_push     :: basic -> bstack -> bstack
```

```
bs_pushB     :: boolean -> bstack -> bstack
bs_pushI     :: int -> bstack -> bstack
bs_update    :: b_dst -> basic -> bstack -> bstack

bs_addI      :: bstack -> bstack
bs_decI      :: bstack -> bstack
bs_eqB       :: bstack -> bstack
bs_eqI       :: bstack -> bstack
bs_eqBi      :: boolean -> b_src -> bstack -> bstack
bs_eqIi      :: int -> b_src -> bstack -> bstack
bs_gtI       :: bstack -> bstack
bs_incI      :: bstack -> bstack
bs_ltI       :: bstack -> bstack
bs_mulI      :: bstack -> bstack
bs_subI      :: bstack -> bstack
```

Definition of the C-stack

```
cs_init      :: cstack
cs_get       :: c_src -> cstack -> instrid
cs_popn      :: c_src -> cstack -> cstack
cs_push      :: instrid -> cstack -> cstack
```

Definition of the Nodes

```
n_arg        :: node -> arg_nr -> arity -> nodeid
n_args       :: node -> arity -> nodeid_seq
n_arity      :: node -> arity
n_B          :: node -> boolean
n_copy       :: node -> node -> node
n_descid     :: node -> descid
n_entry      :: node -> instrid
n_eq_arity   :: node -> arity -> boolean
n_eq_B       :: node -> boolean -> boolean
n_eq_descid  :: node -> descid -> boolean
n_eq_I       :: node -> int -> boolean
n_eq_symbol  :: node -> node -> bool
n_fill       :: descid -> instrid -> args -> node -> node
n_fillB      :: descid -> instrid -> boolean -> node -> node
n_fillI      :: descid -> instrid -> int -> node -> node
n_I          :: node -> int
n_nargs      :: node -> nr_args -> arity -> nodeid_seq
n_setentry   :: instrid -> node -> node
```

Definition of the Graph Store

```
gs_get       :: nodeid -> graphstore -> node
gs_init      :: graphstore
gs_newnode   :: graphstore -> (graphstore,nodeid)
gs_update    :: nodeid -> (node -> node) -> graphstore -> graphstore
```

Definition of the Descriptors

```
d_ap_entry   :: desc -> instrid
d_arity      :: desc -> num
d_name       :: desc -> string
```

Definition of the Descriptor Store

```
ds_get       :: descid -> descstore -> desc
ds_init      :: [desc] -> descstore
```

Definition of the Program Counter

```
pc_init      :: instrid
pc_next      :: instrid -> instrid
pc_halt      :: instrid -> instrid
pc_end       :: instrid -> bool
```

Definition of the Program Store

```
ps_get       :: instrid -> programstore -> instruction
ps_init      :: [instruction] -> programstore
```

Definition of the I/O-channels

```
io_init      :: io
io_print     :: string -> io -> io
```

Micro-instructions to enable a trace

```
show_as      :: astack -> [char]
show_bs      :: bstack -> [char]
show_cs      :: cstack -> [char]
show_io      :: io -> [char]
show_pc      :: instrid -> [char]
show_gs      :: graphstore -> descstore -> [char]
show_nds     :: num -> [node] -> descstore -> [char]
show_nd      :: node -> descstore -> [char]
```

The state of the ABC-machine

```
state        ==  (astack,bstack,cstack,graphstore,descstore,instrid,programstore,io)

instruction  ==  state -> state

ap_entry     ==  instrid
args         ==  nodeid_seq
arity        ==  nat
boolean      ==  bool
entry        ==  instrid
int          ==  num
name         ==  [char]
nat          ==  num
```

```
nodeid       ==   nat
nodeid_seq   ==   [nodeid]
label        ==   [char]
string       ==   [char]

a_dst        ==   nat
a_src        ==   nat
a_src1       ==   nat
a_src2       ==   nat
b_dst        ==   nat
b_src        ==   nat
c_src        ==   nat
nr_args      ==   nat
arg_nr       ==   nat
```

Implementation of the A-stack

```
astack       ==   [nodeid]

as_get target as
 = asget 0 as
   where  asget n (a:x)
            = a                , if n = target
            = asget (n+1) x  , otherwise
          asget n [ ]
            = error ("Taking element "++show target++" of A-stack of length "++show n)

as_init  = [ ]

as_popn n as
 = aspopn n as
   where  aspopn 0 as       =  as
          aspopn m (a:x)   = aspopn (m-1) x
          aspopn m [ ]
          =  error ("Popping "++show n++" elements of A-stack of length "++show(#as))

as_push nodeid as     =  nodeid:as
as_pushn nodeids as =  nodeids ++ as

as_topn 0 as       =  []
as_topn n (x:r)    =  x:(as_topn (n-1) r)
as_topn n [ ]      =  error "as_topn: taking too much elements"

as_update 0 nodeid (a:x) = nodeid:x
as_update n nodeid (a:x) = a : as_update (n-1) nodeid x
as_update n nodeid as    =  error "A-stack update error"
```

Implementation of the B-stack

```
bstack       ==   [basic]
```

```
bs_get target bs
 = bsget 0 bs
   where  bsget n (b:x)
             = b                 , if n = target
             = bsget (n+1) x  , otherwise
          bsget n [ ]
             = error ("Taking element "++show target++" of B-stack of length "++show n)

bs_getB n bs =  b  where  Bool b  =  bs_get n bs
bs_getI n bs  =  i   where  Int i     =  bs_get n bs

bs_copy n bs     = (bs_get n bs):bs

bs_init   = [ ]

bs_popn n bs
 = bspopn n bs
   where  bspopn 0 bs       =  bs
          bspopn m (b:x)   =  bspopn (m-1) x
          bspopn m [ ]
           = error ("Popping "++show n++" elements of B-stack of length "++show(# bs))

bs_push basic bs   =  basic : bs
bs_pushB b bs       =  (Bool b):bs
bs_pushI i bs         =  (Int i):bs

bs_update 0 basic (b:x)   =  basic:x
bs_update n basic (b:x)   =  b : bs_update (n-1) basic x
bs_update n basic bs        =  error "B-stack update error"

bs_addI    ((Int i1):(Int i2):r)         =  (Int (i1+i2)):r
bs_decI    ((Int i1):r)                  =  (Int (i1-1)):r
bs_eqB     ((Bool b1):(Bool b2):r)       =  (Bool (b1=b2)):r
bs_eqBi    b n bs                        =  bs_push (Bool (b=(bs_getB n bs))) bs
bs_eqI     ((Int i1):(Int i2):r)         =  (Bool (i1=i2)):r
bs_eqIi    i n bs                        =  bs_push (Bool (i=(bs_getI n bs))) bs
bs_gtI     ((Int i1):(Int i2):r)         =  (Bool (i1>i2)):r
bs_incI    ((Int i1):r)                  =  (Int (i1+1)):r
bs_ltI     ((Int i1):(Int i2):r)         =  (Bool (i1<i2)):r
bs_mulI    ((Int i1):(Int i2):r)         =  (Int (i1*i2)):r
bs_subI    ((Int i1):(Int i2):r)         =  (Int (i1-i2)):r
```

Implementation of the C-stack

```
cstack          ==    [instrid]

cs_init   = [ ]

cs_get target cs
 = csget 0 cs
   where  csget n (c:x)
             = c                 , if n = target
             = csget (n+1) x  , otherwise
```

```
        csget n [ ]
          =  error ("Taking element "++show target++" of C-stack of length "++show n)

cs_popn n cs
 = cspopn n cs
   where  cspopn 0 cs      =  cs
          cspopn m (c:x)   = cspopn (m-1) x
          cspopn m [ ]
            = error("Popping "++show n++" elements of C-stack of length "++show(#cs))

cs_push c cs  = c : cs
```

Implementation of the Nodes

```
node     ::=  Node    descid instrid args     |
              Basic   descid instrid basic    |
              Empty

basic    ::=  Int     int
              Bool  boolean

n_arg node n arity
 = args !(n-1)                                               , if arity >= n
 = error ("taking arg " ++ show n ++ " from " ++ show node)  , otherwise
   where   args = n_args node arity

n_args (Node     descid entry args) arity
 = args                                                           , if arity = # args
 = error ("wrong arity in n_args: " ++ show arity ++ " (Node " ++
          show descid++" "++show entry++" "++show args++")")    , otherwise
n_args node arity
 = error ("n_args: No args in node: " ++ show node)

n_arity (Basic descid entry basic)   =  0
n_arity (Node descid entry args)     =  # args
n_arity Empty                        =  error "Arity of an empty node is not defined"

n_B (Basic descid entry (Bool b))    =  b
n_B node                             = error   ("N_B: No boolean in node:" ++ show node)

n_copy new old                       =  new

n_descid (Node descid entry args)    =  descid
n_descid (Basic descid entry basic)  =  descid
n_descid Empty                       =  error "No descid in an Empty node!"

n_entry (Node    descid entry args)  =  entry
n_entry (Basic   descid entry basic) =  entry
n_entry Empty                        =  error "No entry in an Empty node!"

n_eq_arity    node n                 =  n_arity node = n
n_eq_B        node b                 =  n_B node = b
n_eq_descid   node descid            =  n_descid node = descid
n_eq_I        node i                 =  n_I node = i
```

```
n_eq_symbol (Node descid1 entry1 args1) (Node descid2 entry2 args2)
                                 = descid1 = descid2
n_eq_symbol (Basic descid1 entry1 basic1) (Basic descid2 entry2 basic2)
                                 = descid1 = descid2 & basic1 = basic2
n_eq_symbol node1 node2          = False

n_fill desc entry args node      = Node desc entry args
n_fillB desc entry b node        = Basic desc entry (Bool b)
n_fillI desc entry i node        = Basic desc entry (Int i)

n_I (Basic descid entry (Int i)) = i
n_I node                         = error ("n_I: No integer in node: " ++ show node)

n_nargs node arg_count arity     = take arg_count (n_args node arity)

n_setentry newentry (Node  descid entry args)  = Node descid newentry args
n_setentry newentry (Basic descid entry basic) = Basic descid newentry basic
n_setentry newentry Empty                      = error "Cannot set entry of Empty node!"
```

Implementation of the Descriptors

```
desc      ::=   Desc ap_entry arity name

d_ap_entry   (Desc ap_entry arity name) = ap_entry
d_arity      (Desc ap_entry arity name) = arity
d_name       (Desc ap_entry arity name) = name
```

Implementation of the Descriptor Store

```
descid      ==  num
descstore   ==  [desc]

ds_get target ds
 = dsget 0 ds
   where dsget n (d:x)
            = d                , if n = target
            = dsget (n+1) x  , otherwise
         dsget n [ ]
            = error ("Taking descriptor "++show target++" of store of size "++show n)

ds_init descriptors  =  descriptors
```

Implementation of the Graph Store

```
graphstore    ==   ([node], nat)

gs_get nodeid (nds,free)
 = gsget (nodeid-free-1) nds
   where gsget 0 (n:ns)   = n
         gsget m (n:ns)   = gsget (m-1) ns
         gsget m [ ]      = error ("unexisting nodeid : " ++ show nodeid)

gs_init  = ([ ], store_size)
```

```
store_size     nat
store_size =  100            || some natural number indicating the size

gs_newnode (nds,0)        =  error "graph-store is full"
gs_newnode (nds,free)     =  ((Empty nds,free-1),free)

gs_update nodeid f (nodes,free)
 = (update place nodes f,free)                            ,if place<=store_size free
 = error ("gs update error  node "++show nodeid++" not existent"),otherwise
   where   place = nodeid - free -1

update  : : nat -> [node] -> (node->node) -> [node]
update 0 (node nodes) f  =  f node nodes
update n (node nodes) f  =  node (update (n-1) nodes f)
```

Implementation of the Program Store

```
programstore     ==    [location]

location              =    I instruction

ps_init (instruction rest)   =  I instruction  ps_init rest
ps_init [ ]                  =  []

ps_get target ps
 = psget 0 ps
   where   psget locus (I instr rest)
              = instr                      , if locus = target
              = psget (locus+1) rest  , otherwise
           psget locus [ ] = error ("Program counter outside program  " ++ show target)
```

Implementation of the Program Counter

```
instrid       ==    num
pc            ==    instrid

pc_end      instrid    = instrid < 0
pc_init          = 0
pc_halt instrid = -1
pc_next instrid = instrid + 1
```

Implementation of the I/O channels

```
io           ==    [char]

io_init       =  []

io_print string output   =  output ++ string

symbol_to_string    node -> desc -> string
symbol_to_string (Basic descid ap_entry (Int i))      desc                     =  show i
symbol_to_string (Basic descid ap_entry (Bool b)) desc                     =  show b
symbol_to_string (Node descid entry args)(Desc ap_entry arity name)     =  name
```

```
show_state : : state -> [char]
show_state (as,bs,cs,gs,ds,pc,ps,io)
 = "output  : " ++ show_io io    ++ "\n" ++
   "pc      : " ++ show_pc pc    ++ "\n" ++
   "A-stack : " ++ show_as as    ++ "\n" ++
   "B-stack : " ++ show_bs bs    ++ "\n" ++
   "C-stack : " ++ show_cs cs    ++ "\n" ++
   "Graph   : (nodeid : node):      "\n" ++  show_gs gs ds

show_as  as          =  show as
show_bs  bs          =  show bs
show_cs  cs          =  show cs
show_io  io          =  io
show_gs (nds,free) ds =  show_nds (free+1) nds ds
show_pc pc           =  show pc                                 , if pc >= 0
                     =  "program is terminated; pc undefined"   , otherwise
show_nds i (n:nds) ds =  show i++" : "++show_nd n ds++"\n"++show_nds (i+1) nds ds
show_nds i [ ] ds    =  [ ]
show_nd (Node  descid entry args) ds
                     =  d_name (ds_get descid ds)++ " "++show entry++" "++show args
show_nd (Basic descid entry basic) ds  =  show entry ++ " " ++ show basic
show_nd Empty ds     =  "Empty"

cond : : bool -> * -> * -> *
cond b then else
 = then, if b
 = else, otherwise
```

A.2 The instruction set

Type definitions of the instructions

```
add_args        :: a_src -> nr_args -> a_dst -> instruction
create          :: instruction
del_args        :: a_src -> nr_args -> a_dst -> instruction
dump            :: string -> instruction
eq_desc         :: descid -> a_src -> instruction
eq_desc_arity   :: descid -> arity -> a_src -> instruction
eq_symbol       :: a_src1 -> a_src2 -> instruction
eqB             :: instruction
eqB_a           :: bool -> a_src -> instruction
eqB_b           :: bool -> b_src -> instruction
eqI             :: instruction
eqI_a           :: int -> a_src -> instruction
eqI_b           :: int -> b_src -> instruction
fill            :: descid -> nr_args -> instrid -> a_dst -> instruction
fill_a          :: a_src -> a_dst -> instruction
fillB           :: bool -> a_dst -> instruction
```

```
fillB_b         :: b_src -> a_dst -> instruction
fillI           :: int     -> a_dst -> instruction
fillI_b         :: b_src -> a_dst -> instruction
get_desc_arity  :: a_src -> instruction
get_node_arity  :: a_src -> instruction
halt            :: instruction
jmp             :: instrid -> instruction
jmp_eval        :: instruction
jmp_false       :: instrid -> instruction
jmp_true        :: instrid -> instruction
jsr             :: instrid -> instruction
jsr_eval        :: instruction
no_op           :: instruction
pop_a           :: nr_args -> instruction
pop_b           :: nr_args -> instruction
print           :: string -> instruction
print_symbol    :: a_src -> instruction
push_a          :: a_src -> instruction
push_ap_entry   :: a_src -> instruction
push_arg        :: a_src -> arity -> arg_nr -> instruction
push_arg_b      :: a_src -> instruction
push_args       :: a_src -> arity -> nr_args -> instruction
push_args_b     :: a_src -> instruction
push_b          :: b_src -> instruction
pushB           :: bool -> instruction
pushB_a         :: a_src -> instruction
pushI           :: int -> instruction
pushI_a         :: a_src -> instruction
repl_args       :: arity -> nr_args -> instruction
repl_args_b     :: instruction
rtn             :: instruction
set_entry       :: instrid -> a_dst -> instruction
update_a        :: a_src -> a_dst -> instruction
update_b        :: b_src -> b_dst -> instruction

addI            :: instruction
decI            :: instruction
gtI             :: instruction
incI            :: instruction
ltI             :: instruction
mulI            :: instruction
subI            :: instruction
```

Implementation of the instructions

```
add_args a_src nr_args a_dst (as,bs,cs,gs,ds,pc,ps,io)
= (as',bs,cs,gs',ds,pc,ps,io)
  where as'       = as_popn nr_args as
        gs'       = gs_update dstid (n_fill descid entry newargs) gs
        dstid     = as_get a_dst as
        srcid     = as_get a_src as
        node      = gs_get srcid gs
        descid    = n_descid node
        entry     = n_entry node
        arity     = n_arity node
        newargs   = n_args node arity ++ as_topn nr_args as

create (as,bs,cs,gs,ds,pc,ps,io)
= (as',bs,cs,gs',ds,pc,ps,io)
  where as'           = as_push nodeid as
        (gs',nodeid)  = gs_newnode gs

del_args a_src nr_args a_dst (as,bs,cs,gs,ds,pc,ps,io)
= (as',bs,cs,gs',ds,pc,ps,io)
  where as'       = as_pushn newargs as
        gs'       = gs_update dstid (n_fill descid entry newargs) gs
        dstid     = as_get a_dst as
        srcid     = as_get a_src as
        node      = gs_get srcid gs
        descid    = n_descid node
        entry     = n_entry node
        newargs   = n_nargs node (arity-nr_args) arity
        arity     = n_arity node

dump string (as,bs,cs,gs,ds,pc,ps,io)
= (as,bs,cs,gs,ds,pc,ps,io')
  where io'       = o_print ("\n"++string++"\n"++state) io
        state     = show_state (as,bs,cs,gs,ds,pc,ps,io)

eq_desc descid a_src (as,bs,cs,gs,ds,pc,ps,io)
= (as,bs',cs,gs,ds,pc,ps,io)
  where bs'       = bs_pushB equal bs
        equal     = n_eq_descid node descid
        node      = gs_get nodeid gs
        nodeid    = as_get a_src as

eq_desc_arity descid arity a_src (as,bs,cs,gs,ds,pc,ps,io)
= (as,bs',cs,gs,ds,pc,ps,io)
  where bs'       = bs_pushB equal bs
        equal     =  (n_eq_descid node descid) & (n_eq_arity node arity)
        node      = gs_get nodeid gs
        nodeid    = as_get a_src as
```

```
eq_symbol a_src1 a_src2 (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'          = bs_pushB equal bs
          equal        = n_eq_symbol node1 node2
          node1        = gs_get nodeid1 gs
          node2        = gs_get nodeid2 gs
          nodeid1      = as_get a_src1 as
          nodeid2      = as_get a_src2 as

eqB (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'          = bs_eqB bs

eqB_a bool a_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'          = bs_pushB equal bs
          equal        = n_eq_B (gs_get nodeid gs) bool
          nodeid       = as_get a_src as

eqB_b bool b_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'          = bs_eqBi bool b_src bs

eqI (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'          = bs_eqI bs

eqI_a int a_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'          = bs_pushB equal bs
          equal        = n_eq_I (gs_get nodeid gs) int
          nodeid       = as_get a_src as

eqI_b int b_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'          = bs_eqIi int b_src bs

fill desc nr_args entry a_dst (as,bs,cs,gs,ds,pc,ps,io)
 = (as',bs,cs,gs',ds,pc,ps,io)
   where  as'          = as_popn nr_args as
          gs'          = gs_update nodeid (n_fill desc entry args) gs
          nodeid       = as_get a_dst as
          args         = as_topn nr_args as

fill_a a_src a_dst (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs,gs',ds,pc,ps,io)
   where  gs'            = gs_update nodeid_dst (n_copy node_src) gs
          node_src       = gs_get nodeid_src gs
          nodeid_dst     = as_get a_dst as
          nodeid_src     = as_get a_src as
```

```
fillB bool a_dst (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs,gs',ds,pc,ps,io)
   where  gs'     = gs_update nodeid (n_fillB bool_desc rnf_entry bool) gs
          nodeid  = as_get a_dst as

fillB_b b_src a_dst (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs,gs',ds,pc,ps,io)
   where  gs'     = gs_update nodeid (n_fillB bool_desc rnf_entry bool) gs
          bool    = bs_getB b_src bs
          nodeid  = as_get a_dst as

fillI int a_dst (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs,gs',ds,pc,ps,io)
   where  gs'     = gs_update nodeid (n_fillI int_desc rnf_entry int) gs
          nodeid  = as_get a_dst as

fillI_b b_src a_dst (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs,gs',ds,pc,ps,io)
   where  gs'     = gs_update nodeid (n_fillI int_desc rnf_entry int) gs
          int     = bs_getI b_src bs
          nodeid  = as_get a_dst as

get_desc_arity a_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'     = bs_pushI arity bs
          arity   = d_arity (ds_get descid ds)
          descid  = n_descid (gs_get nodeid gs)
          nodeid  = as_get a_src as

get_node_arity a_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'     = bs_pushI arity bs
          arity   = n_arity (gs_get nodeid gs)
          nodeid  = as_get a_src as

halt (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs,gs,ds,pc',ps,io)
   where  pc'     = pc_halt pc

jmp address (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs,gs,ds,pc',ps,io)
   where  pc'     = address

jmp_eval (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs,gs,ds,pc',ps,io)
   where  pc'     = n_entry (gs_get nodeid gs)
          nodeid  = as_get 0 as
```

```
jmp_false address (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc',ps,io)
   where  pc'      =  cond (~bool) address pc
          bool     =  bs_getB 0 bs
          bs'      =  bs_popn 1 bs

jmp_true address (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc',ps,io)
   where  pc'      =  cond bool address pc
          bool     =  bs_getB 0 bs
          bs'      =  bs_popn 1 bs

jsr address (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs',gs,ds,pc',ps,io)
   where  pc'      =  address
          cs'      =  cs_push pc cs

jsr_eval (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs',gs,ds,pc',ps,io)
   where  pc'      =  n_entry (gs_get nodeid gs)
          nodeid   =  as_get 0 as
          cs'      =  cs_push pc cs

no_op (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs,gs,ds,pc,ps,io)

pop_a n (as,bs,cs,gs,ds,pc,ps,io)
 = (as',bs,cs,gs,ds,pc,ps,io)
   where  as'      =  as_popn n as

pop_b n (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'      =  bs_popn n bs

print string (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs,gs,ds,pc,ps,io')
   where  io'      =  io_print string io

print_symbol a_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs,gs,ds,pc,ps,io')
   where  io'      =  io_print string io
          nodeid   =  as_get a_src as
          node     =  gs_get nodeid gs
          string   =  symbol_to_string node desc
          desc     =  ds_get (n_descid node) ds

push_a a_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as',bs,cs,gs,ds,pc,ps,io)
   where  as'      =  as_push nodeid as
          nodeid   =  as_get a_src as
```

```
push_ap_entry a_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs',gs,ds,pc,ps,io)
   where cs'     = cs_push (d_ap_entry (ds_get descid ds)) cs
         descid  = n_descid (gs_get nodeid gs)
         nodeid  = as_get a_src as

push_arg a_src arity arg_nr (as,bs,cs,gs,ds,pc,ps,io)
 = (as',bs,cs,gs,ds,pc,ps,io)
   where as'     = as_push arg as
         arg     = n_arg (gs_get nodeid gs) arg_nr arity
         nodeid  = as_get a_src as

push_arg_b a_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as',bs,cs,gs,ds,pc,ps,io)
   where as'     = as_push arg as
         arg     = n_arg (gs_get nodeid gs) arg_nr arity
         nodeid  = as_get a_src as
         arg_nr  = bs_getI 0 bs
         arity   = bs_getI 1 bs

push_args a_src arity nr_args (as,bs,cs,gs,ds,pc,ps,io)
 = (as',bs,cs,gs,ds,pc,ps,io)
   where as'     = as_pushn args as
         args    = n_nargs (gs_get nodeid gs) nr_args arity
         nodeid  = as_get a_src as

push_args_b a_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as',bs,cs,gs,ds,pc,ps,io)
   where as'     = as_pushn args as
         args    = n_nargs (gs_get nodeid gs) nargs arity
         nargs   = bs_getI 0 bs
         nodeid  = as_get a_src as
         arity   = bs_getI 1 bs

push_b b_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where bs'     = bs_push basic bs
         basic   = bs_get b_src bs

pushB bool (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where bs'     = bs_pushB bool bs

pushB_a a_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where bs'     = bs_pushB bool bs
         bool    = n_B (gs_get nodeid gs)
         nodeid  = as_get a_src as

pushI int (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where bs'     = bs_pushI int bs
```

```
pushI_a a_src (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'      =  bs_pushI int bs
          int      =  n_I (gs_get nodeid gs)
          nodeid   =  as_get a_src as

repl_args arity nr_args (as,bs,cs,gs,ds,pc,ps,io)
 = (as',bs,cs,gs,ds,pc,ps,io)
   where  as'      =  as_pushn args (as_popn 1 as)
          args     =  n_nargs (gs_get nodeid gs) nr_args arity
          nodeid   =  as_get 0 as

repl_args_b (as,bs,cs,gs,ds,pc,ps,io)
 = (as',bs,cs,gs,ds,pc,ps,io)
   where  as'      =  as_pushn args (as_popn 1 as)
          args     =  n_nargs (gs_get nodeid gs) nr_args arity
          nodeid   =  as_get 0 as
          arity    =  bs_getI 0 bs
          nr_args  =  bs_getI 1 bs

rtn (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs',gs,ds,pc',ps,io)
   where  pc'      =  cs_get 0 cs
          cs'      =  cs_popn 1 cs

set_entry entry a_dst (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs,gs',ds,pc,ps,io)
   where  gs'      =  gs_update nodeid (n_setentry entry) gs
          nodeid   =  as_get a_dst as

update_a a_src a_dst (as,bs,cs,gs,ds,pc,ps,io)
 = (as',bs,cs,gs,ds,pc,ps,io)
   where  as'      =  as_update a_dst nodeid as
          nodeid   =  as_get a_src as

update_b b_src b_dst (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'      =  bs_update b_dst basic bs
          basic    =  bs_get b_src bs

addI (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'      =  bs_addI bs

decI (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'      =  bs_decI bs

gtI (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'      =  bs_gtI bs
```

```
incI (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'      = bs_incI bs

ltI (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'      = bs_ltI bs

mulI (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'      = bs_mulI bs

subI (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs',cs,gs,ds,pc,ps,io)
   where  bs'      = bs_subI bs
```

The driver

```
boot : : ([instruction],[desc]) -> state
boot (program,descriptors)
 = (as,bs,cs,gs,ds,pc,ps,io)
   where  pc   = pc_init
          as   = as_init
          bs   = bs_init
          cs   = cs_init
          gs   = gs_init
          ps   = ps_init program
          io   = Io_init
          ds   = ds_init descriptors

fetch_cycle : : state -> state
fetch_cycle (as,bs,cs,gs,ds,pc,ps,io)
 = (as,bs,cs,gs,ds,pc,ps,io), pc_end pc
 = fetch_cycle (currinstr (as,bs,cs,gs,ds,pc',ps,io)), otherwise
   where  pc'       = pc_next pc
          currinstr = ps_get pc ps

step_cycle (as,bs,cs,gs,ds,pc,ps,io)
 = show_state (as,bs,cs,gs,ds,pc,ps,io), pc_end pc
 = show_state (as,bs,cs,gs,ds,pc,ps,io) ++
   step_cycle (currinstr (as,bs,cs,gs,ds,pc',ps,io')), otherwise
   where  pc'       = pc_next pc
          currinstr = ps_get pc ps
          io'       = Io_init

int_desc   = 0
bool_desc  = 1
rnf_entry  = 1
```

A.3 ABC-assembler

ABC-assembler is represented by an AST. The mapping from ABC-assembler to ABC-instructions is a simple abstract program transformation.

```
assembler     ==   [statement]
red_label     ==   label
desc_label    ==   label
nr_instr      ==   int

statement
   ::=   Label            label                              |
         Descriptor       desc_label red_label arity name    |
         Br               nr_instr                           |
         Br_false         nr_instr                           |
         Br_true          nr_instr                           |
         Dump             string                             |
         Add_args         a_src nr_args a_dst                |
         Create                                              |
         Del_args         a_src nr_args a_dst                |
         Eq_desc          desc_label a_src                   |
         Eq_desc_arity    desc_label arity a_src             |
         EqB                                                 |
         EqB_a            bool a_src                         |
         EqB_b            bool b_src                         |
         EqI                                                 |
         EqI_a            int a_src                          |
         EqI_b            int b_src                          |
         Fill             desc_label nr_args label a_dst     |
         Fill_a           a_src a_dst                        |
         FillB            bool a_dst                         |
         FillB_b          b_src a_dst                        |
         FillI            int a_dst                          |
         FillI_b          b_src a_dst                        |
         Get_desc_arity   a_src                              |
         Get_node_arity   a_src                              |
         Halt                                                |
         Jmp              label                              |
         Jmp_eval                                            |
         Jmp_false        label                              |
         Jmp_true         label                              |
         Jsr              label                              |
         Jsr_eval                                            |
         No_op                                               |
         Pop_a            nat                                |
         Pop_b            nat                                |
          Print           string                        |
          Print_symbol    a_src                         |
          Push_a          a_src                         |
          Push_ap_entry   a_src                         |
          Push_arg        a_src arity arg_nr            |
          Push_arg_b      a_src                         |
          Push_args       a_src arity arg_nr            |
          Push_args_b     a_src                         |
          Push_b          nat                           |
          PushB           bool                          |
          PushB_a         a_src                         |
          PushI           int                           |
          PushI_a         a_src                         |
          Repl_args       arity nr_args                 |
          Repl_args_b                                   |
          Rtn                                           |
          Set_entry       label a_dst                   |
          Update_a        a_src a_dst                   |
          Update_b        b_src b_dst                   |

          AddI                                          |
          DecI                                          |
          GtI                                           |
          IncI                                          |
          LtI                                           |
          MulI                                          |
          SubI

assemble : : assembler -> ([instruction],[desc])
assemble statements
 = (translate statements loc_counter sym_table,desc_table statements sym_table)
   where  loc_counter      =  0
          desc_counter     =  0
          sym_table        =  collect statements loc_counter desc_counter

collect : : assembler -> nat -> nat -> sym_table
collect (Label l:r) lc dc      =  (l,lc,Lab_sym):collect r lc dc
collect (Descriptor desc_label red_label arity name:r) lc dc
                               =  (desc_label,dc,Desc_sym):collect r lc (dc+1)
collect (instr:r) lc dc        =  collect r (lc+1) dc
collect [ ] lc dc              =  [ ]

look_up : : label -> sym_type -> sym_table -> nat
look_up lab t ((name,n,sym_type):r)
 = n                     , if lab = name & t = sym_type
 = look_up lab t r       , otherwise
look_up lab t [ ]  =  error ("\n\nlabel " ++ lab ++ " not defined as " ++ show t ++ "\n")

sym_table  ==   [(name,nat,sym_type)]
sym_type   ::=  Lab_sym   |   Desc_sym
```

```
desc_table: :assembler -> sym_table -> [desc]
desc_table (Descriptor desc_label ap_entry arity name:r) sym_table
 = (Desc ap_addr arity name):desc_table r sym_table
   where   ap_addr = look_up ap_entry Lab_sym sym_table
desc_table (instr:r) sym_table  =  desc_table r sym_table
desc_table [ ] sym_table        =  [ ]

translate : :assembler -> nat -> sym_table -> [instruction]
translate (Label name                          :           r)  lc       sym_table
 =                                               translate r   lc       sym_table
translate (Descriptor lab ap_entry arity name  :           r)  lc       sym_table
 =                                               translate r   lc       sym_table
translate (Br n                                :           r)  lc       sym_table
 = jmp (lc+n+1)                                : translate r   (lc+1)   sym_table
translate (Br_false n                          :           r)  lc       sym_table
 = jmp_false (lc+n+1)                          : translate r   (lc+1)   sym_table
translate (Br_true n                           :           r)  lc       sym_table
 = jmp_true (lc+n+1)                           : translate r   (lc+1)   sym_table
translate (Dump string                         :           r)  lc       sym_table
 = dump string                                 : translate r   (lc+1)   sym_table
translate (Add_args a_src n a_dst              :           r)  lc       sym_table
 = add_args a_src n a_dst                      : translate r   (lc+1)   sym_table
translate (Create                              :           r)  lc       sym_table
 = create                                      : translate r   (lc+1)   sym_table
translate (Del_args a_src n a_dst              :           r)  lc       sym_table
 = del_args a_src n a_dst                      : translate r   (lc+1)   sym_table
translate (Eq_desc desc_label a_src            :           r)  lc       sym_table
 = eq_desc desc_addr a_src                     : translate r   (lc+1)   sym_table
   where   desc_addr    = look_up desc_label Desc_sym sym_table
translate (Eq_desc_arity desc_label arity a_src :          r)  lc       sym_table
 = eq_desc_arity desc_addr arity a_src         : translate r   (lc+1)   sym_table
   where   desc_addr    = look_up desc_label Desc_sym sym_table
translate (EqB                                 :           r)  lc       sym_table
 = eqB                                         : translate r   (lc+1)   sym_table
translate (EqB_a bool a_src                    :           r)  lc       sym_table
 = eqB_a bool a_src                            : translate r   (lc+1)   sym_table
translate (EqB_b bool b_src                    :           r)  lc       sym_table
 = eqB_b bool b_src                            : translate r   (lc+1)   sym_table
translate (EqI                                 :           r)  lc       sym_table
 = eqI                                         : translate r   (lc+1)   sym_table
translate (EqI_a int a_src                     :           r)  lc       sym_table
 = eqI_a int a_src                             : translate r   (lc+1)   sym_table
translate (EqI_b int b_src                     :           r)  lc       sym_table
 = eqI_b int b_src                             : translate r   (lc+1)   sym_table
translate (Fill desc_lab nr_args entry_lab a_dst :         r)  lc       sym_table
 = fill desc_addr nr_args entry_addr a_dst     : translate r   (lc+1)   sym_table
   where   desc_addr    = look_up desc_lab Desc_sym sym_table
           entry_addr   = look_up entry_lab Lab_sym sym_table
```

```
translate (Fill_a a_src a_dst                 :             r)  lc       sym_table
 = fill_a a_src a_dst                         : translate  r   (lc+1)   sym_table
translate (FillB bool a_dst                   :             r)  lc       sym_table
 = fillB bool a_dst                           : translate  r   (lc+1)   sym_table
translate (FillB_b b_src a_dst                :             r)  lc       sym_table
 = fillB_b b_src a_dst                        : translate  r   (lc+1)   sym_table
translate (FillI int a_dst                    :             r)  lc       sym_table
 = fillI int a_dst                            : translate  r   (lc+1)   sym_table
translate (FillI_b b_src a_dst                :             r)  lc       sym_table
 = fillI_b b_src a_dst                        : translate  r   (lc+1)   sym_table
translate (Get_desc_arity a_src               :             r)  lc       sym_table
 = get_desc_arity a_src                       : translate  r   (lc+1)   sym_table
translate (Get_node_arity a_src               :             r)  lc       sym_table
 = get_node_arity a_src                       : translate  r   (lc+1)   sym_table
translate (Halt                               :             r)  lc       sym_table
 = halt                                       : translate  r   (lc+1)   sym_table
translate (Jmp label                          :             r)  lc       sym_table
 = jmp address                                : translate  r   (lc+1)   sym_table
   where  address    =  look_up label Lab_sym sym_table
translate (Jmp_eval                           :             r)  lc       sym_table
 = jmp_eval                                   : translate  r   (lc+1)   sym_table
translate (Jmp_false label                    :             r)  lc       sym_table
 = jmp_false address                          : translate  r   (lc+1)   sym_table
   where  address    =  look_up label Lab_sym sym_table
translate (Jmp_true label                     :             r)  lc       sym_table
 = jmp_true address                           : translate  r   (lc+1)   sym_table
   where  address    =  look_up label Lab_sym sym_table:
translate (Jsr label                          :             r)  lc       sym_table
 = jsr address                                : translate  r   (lc+1)   sym_table
   where  address    =  look_up label Lab_sym sym_table
translate (Jsr_eval                           :             r)  lc       sym_table
 = jsr_eval                                   : translate  r   (lc+1)   sym_table
translate (No_op                              :             r)  lc       sym_table
 = no_op                                      : translate  r   (lc+1)   sym_table
translate (Pop_a nat                          :             r)  lc       sym_table
 = pop_a nat                                  : translate  r   (lc+1)   sym_table
translate (Pop_b nat                          :             r)  lc       sym_table
 = pop_b nat                                  : translate  r   (lc+1)   sym_table
translate (Print string                       :             r)  lc       sym_table
 = print string                               : translate  r   (lc+1)   sym_table
translate (Print_symbol a_src                 :             r)  lc       sym_table
 = print_symbol a_src                         : translate  r   (lc+1)   sym_table
translate (Push_a a_src                       :             r)  lc       sym_table
 = push_a a_src                               : translate  r   (lc+1)   sym_table
translate (Push_ap_entry a_src                :             r)  lc       sym_table
 = push_ap_entry a_src                        : translate  r   (lc+1)   sym_table
translate (Push_arg a_src arity arg_nr        :             r)  lc       sym_table
 = push_arg a_src arity arg_nr                : translate  r   (lc+1)   sym_table
```

```
translate (Push_arg_b a_src                 :           r) lc     sym_table
 = push_arg_b a_src                         : translate r  (lc+1) sym_table
translate (Push_args a_src arity nr_args    :           r) lc     sym_table
 = push_args a_src arity nr_args            : translate r  (lc+1) sym_table
translate (Push_args_b a_src                :           r) lc     sym_table
 = push_args_b a_src                        : translate r  (lc+1) sym_table
translate (Push_b nat                       :           r) lc     sym_table
 = push_b nat                               : translate r  (lc+1) sym_table
translate (PushB bool                       :           r) lc     sym_table
 = pushB bool                               : translate r  (lc+1) sym_table
translate (PushB_a a_src                    :           r) lc     sym_table
 = pushB_a a_src                            : translate r  (lc+1) sym_table
translate (PushI int                        :           r) lc     sym_table
 = pushI int                                : translate r  (lc+1) sym_table
translate (PushI_a a_src                    :           r) lc     sym_table
 = pushI_a a_src                            : translate r  (lc+1) sym_table
translate (Repl_args arity nr_args          :           r) lc     sym_table
 = repl_args arity nr_args                  : translate r  (lc+1) sym_table
translate (Repl_args_b                      :           r) lc     sym_table
 = repl_args_b                              : translate r  (lc+1) sym_table
translate (Rtn                              :           r) lc     sym_table
 = rtn                                      : translate r  (lc+1) sym_table
translate (Set_entry label a_dst            :           r) lc     sym_table
 = set_entry address a_dst                  : translate r  (lc+1) sym_table
   where  address   = look_up label Lab_sym sym_table
translate (Update_a a_src a_dst             :           r) lc     sym_table
 = update_a a_src a_dst                     : translate r  (lc+1) sym_table
translate (Update_b a_src a_dst             :           r) lc     sym_table
 = update_b a_src a_dst                     : translate r  (lc+1) sym_table
translate (AddI                             :           r) lc     sym_table
 = addI                                     : translate r  (lc+1) sym_table
translate (DecI                             :           r) lc     sym_table
 = decI                                     : translate r  (lc+1) sym_table
translate (GtI                              :           r) lc     sym_table
 = gtI                                      : translate r  (lc+1) sym_table
translate (IncI                             :           r) lc     sym_table
 = incI                                     : translate r  (lc+1) sym_table
translate (LtI                              :           r) lc     sym_table
 = ltI                                      : translate r  (lc+1) sym_table
translate (MulI                             :           r) lc     sym_table
 = mulI                                     : translate r  (lc+1) sym_table
translate (SubI                             :           r) lc     sym_table
 = subI                                     : translate r  (lc+1) sym_table
translate (x                                :           r) lc     sym_table
 = error ("\n\nUnknown assembler statemant: " ++ show x )
translate [ ] lc sym_table = [ ]
```

```
listing : : assembler -> [char]
listing statements
 = list statements loc_counter sym_table
   where  loc_counter   =  0
          desc_counter =  0
          sym_table     =  collect statements loc_counter desc_counter

list : : assembler -> nat -> sym_table -> [char]
list (Label name                                          r)  lc       sym_table
 = "\n\t"++name ++ ":" ++ list r   lc  sym_table
list (Descriptor lab ap_entry arity name           :          r)  lc       sym_table
 = "\n\t"++lab++"("++desc_num++"):\tDescriptor "++ap_entry++
   " ("++ ap_addr++") "++show arity++" "++name          ++    list r  lc       sym_ table
   where  ap_addr       =  show (look_up ap_entry Lab_sym sym_table)
          desc_num      =  show (look_up lab Desc_sym sym_table)
list (Br n                                         :          r)  lc       sym_table
 = ln lc++"||Br "++show n++"\n\t\tJmp "++show (lc+n+1)  ++    list    r  (lc+1)  sym_table
list (Br_false n                                   :          r)  lc       sym_table
 = ln lc ++ "||Br_false " ++ show n ++ "\n\t\tJmp_false "
   ++ show (lc+n+1)                                ++    list    r  (lc+1)  sym_table
list (Br_true n                                    :          r)  lc       sym_table
 = ln lc ++ "||Br_true " ++ show n ++ "\n\t\tJmp_true "
   ++ show (lc+n+1)                                ++    list    r  (lc+1)  sym_table
list (Eq_desc desc_label a_src                     :          r)  lc       sym_table
 = ln lc++"Eq_desc "++desc_label++" ("++
   show desc_addr++") "++show a_src                ++    list    r  (lc+1)  sym_table
   where   desc_addr     =  ook_up desc_label Desc_sym sym_table
list (Eq_desc_arity desc_label arity a_src         :          r)  lc       sym_table
 = ln lc++"Eq_desc_arity "++desc_label++" ("++desc_addr
   ++") "++show arity++" "++show a_src             ++    list    r  (lc+1)  sym_table
   where   desc_addr     =  show (look_up desc_label Desc_sym sym_table)
list (Fill desc_lab nr_args entry_lab a_dst  :     :          r)  lc       sym  table
 = ln lc++"Fill "++desc_lab++" ("++desc_addr++") "++ show nr_args++
   " "++entry_lab++" ("++ entry_addr++") "++show a_dst ++   list    r  (lc+1)  sym_table
   where   desc_addr     =  show (look_up desc_lab Desc_sym  sym_table)
           entry_addr    =  show (look_up entry_lab Lab_sym   sym_table)
list (Jmp label                                    :          r)  lc       sym_table
 = ln lc ++ "Jmp " ++ label ++ " (" ++ show address ++")"  ++    list    r  (lc+1)  sym_table
   where   address = look_up label Lab_sym sym_table
list (Jmp_false label                              :          r)  lc       sym_table
 = ln lc++"Jmp_false "++label++" ("++show address++")"++    list    r  (lc+1)  sym_table
   where   address = look_up label Lab_sym sym_table
list (Jmp_true label                               :          r)  lc       sym_table
 = ln lc++"Jmp_true "++label++" ("++ show address ++ ")"++   list    r  (lc+1)  sym_table
   where   address = look_up label Lab_sym sym_table
list (Jsr label                                    :          r)  lc       sym_table
 = ln lc ++ "Jsr " ++ label ++ " ("++ show address ++ ")"   ++    list    r  (lc+1)  sym_table
   where   address     =  look_up label Lab_sym sym_table
```

```
list (Set_entry label a_dst                                 r) lc     sym_table
 = ln lc++"Set_entry "++label++
   " ("++show address++")"++show a_dst            ++   list r  (lc+1) sym_table
   where   address = look_up label Lab_sym sym_table
list (instr                                       :         r) lc     sym_table
 = ln lc ++ show instr                            ++   list r  (lc+1) sym_table
list [ ] lc sym_table
 = []

ln : : num -> [char]
ln lc = "\n" ++ show lc ++ "\t\t"
```

Appendix B
Translating Clean to ABC-code

A complete and executable specification of the translations of Clean to ABC-code is given in this appendix. The translation is explained and illustrated with a number of examples in chapter 5. The Clean program is represented by the AST defined in section 5.3. The generated ABC-assembler is stored in thc AST defined in appendix A. The machine specification in given in that chapter can be used to execute the translated Clean programs.

```
bindings  ==  node_id -> offset   || Associates an index in the A-stack frame to a node-id
offset    ==  num                 || An index in the A-stack frame
asp       ==  num
state     ==  node_id -> status   || Associates the status of the node to a node-id
status                            || The states possible
          ::= Undefined         | || This node-id is not defined in the current scope
              Created           | || The node is created but is still empty
              Unknown           | || The node is filled; it is unknown if the graph is a rnf
              InRnf               || This node is in root normal form

compiler : : clean -> abc_assembler
compiler      = concat.map compile

compile :: rewrite_rule -> abc_assembler
compile (TypedRule typealts alts)
 = fun_descriptor lhs  ++
   prepare_args lhs    ++
   alt_entries alts 1  ++
   gen_type_error symbol_id (# args) (# alts)
   where  lhs = get_lhs (hd typealts)
```

```
            NODE annots nodeid (Function symbol_id) args = hd lhs
compile (ConsRule typealts alts)
 = fun_descriptor lhs  ++
   prepare_args lhs    ++
   alt_entries alts 1  ++
   fill_with_rnf symbol_id (# args) (# alts)
   where   lhs = get_lhs (hd typealts)
            NODE annots nodeid (Function symbol_id) args = hd lhs
compile (UntypedRule alts)
 =  fun_descriptor lhs ++
   prepare_args lhs    ++
   alt_entries alts 1  ++
   fill_with_rnf symbol_id (# args) (# alts)
   where   lhs = get_lhs (hd alts)
            NODE annots nodeid (Function symbol_id) args = hd lhs
compile (TypeRule typealts)
 = descriptors typealts

fun_descriptor :: graph -> abc_assembler
fun_descriptor (NODE annots nodeid (Function symbol_id) args:r)
 = [  Descriptor
            symbol_id
            (apply_label symbol_id)
            (# args)
            symbol_id                                  ]

descriptors :: rule -> abc_assembler
descriptors (alt:rest)
 = cons_descriptor rhs   ++
   fill_code rhs         ++
   descriptors rest
   where   rhs    =  get_rhs alt
descriptors [ ]
 = [ ]

cons_descriptor :: graph -> abc_assembler
cons_descriptor (NODE annots nodeid (Constructor symbol_id) args:r)
 = [  Descriptor
            symbol_id
            apply_lab
            arity
            symbol_id                                  ]
   where   apply_lab
                =  rnf_lab                                 , if arity = 0
                =  apply_label symbol_id                   , otherwise
            arity  =  # args
cons_descriptor (NODE annots nodeid (Function symbol_id) args:r)
 = [ ]
```

```
fill_code   graph -> abc_assembler
fill_code (NODE annots nodeid (Constructor symbol_id) args r)
 = [  Label       (apply_label symbol_id)                ,
      Fill        symbol_id arity rnf_lab arity          ,
      Rtn                                                ]   , if arity > 0
 = []                                                        , otherwise
   where   arity  =  # args
fill_code (NODE annots nodeid (Function symbol_id) args r)
 = []

prepare_args   graph -> abc_assembler
prepare_args (NODE annots nodeid (Function symbol_id) args  alts)
 = [  Label       (node_label symbol_id)                 ,
      Set_entry   cycle_lab 0                            ]  ++
   pushargs 0 arity                                         ++
   [  Label       (apply_label symbol_id)                ]  ++
   reduce_strict_args args 0
   where   arity  =  # args

reduce_strict_args   arg's -> a_src -> abc_assembler
reduce_strict_args (arg args)   n
 = reduce_arg n ++     reduce_strict_args args (n+1)          , if is_strict arg
 =                     reduce_strict_args args (n+1)          , otherwise
reduce_strict_args [ ]            n  =  []

reduce_arg   a_src -> abc_assembler
reduce_arg a_src
 = [  Jsr_eval                                           ]   , if a_src = 0
 = [  Push_a        a_src                                ,
      Jsr_eval                                           ,
      Pop_a       1                                      ]   , otherwise

fill_with_rnf   symbolid -> arity -> num -> abc_assembler
fill_with_rnf symbol_id arity nr_alts
 = []                                                        , if arity = 0
 = [  Label       (alt_label symbol_id (nr_alts+1))      ,
      Fill        symbol_id arity rnf_lab arity          ,
      Rtn                                                ]   , otherwise

gen_type_error   symbolid -> arity -> num -> abc_assembler
gen_type_error symbol_id arity nr_alts
 = []                                                        , if arity = 0
 = [  Label       (alt_label symbol_id (nr_alts+1))      ,
      Jmp         type_error                             ]   , otherwise
```

B.1 Reduction entries

```
alt_entries   rule -> num -> abc_assembler
alt_entries (alt r) alt_number = alt_entry alt alt_number++alt_entries r (alt_number+1)
alt_entries [ ]      alt_number = [ ]
```

```
alt_entry :: rulealt -> num -> abc_assembler
alt_entry (Rewrite lhs rhs) alt_number
 = match_code ++ contractum rhs bindings states asp
   where   (match_code,bindings,states,asp)    =  match lhs alt_number
alt_entry (Redirect lhs annots nodeid) alt_number
 = match_code ++ redirection annots nodeid bindings states asp
   where   (match_code,bindings,states,asp)    =  match lhs alt_number
```

B.2 Code for the left handside of a rule alternative

```
match :: graph -> num -> (abc_assembler,bindings,states,asp)
match [NODE annots nodeid (Function symbol_id) args] alt_number
 = (code, bindings', states', asp')
   where   code        = [ Label      label     ]  ++    match_code
           bindings    = bind_args args asp empty_bindings
           states      = record_args args initial_status
           asp         = arity
           arity       = # args
           label       = alt_label symbol_id alt_number
           next_alt    = alt_label symbol_id (alt_number+1)
           (match_code,bindings',states',asp')
                       = match_args args bindings states asp next_alt asp arity

match_args :: arg's -> bindings -> states -> asp -> label -> offset -> arity ->
                                                  (abc_assembler,bindings,states,asp)
match_args (NodeId an nodeid:rest) bindings states asp next offset arity
 = match_args rest bindings states asp next (offset-1) arity
match_args (Term ans (NODE n_ans id sym args):rest) binds states asp next offset arity
 = (code, bindings", states", asp")
   where   code     = bring_in_rnf annots a_src   ++    match_code  ++    match_rest
           a_src    = asp - offset
           (match_rest,bindings",states",asp")
                    = match_args rest bindings' states' asp' next (offset-1) arity
           (match_code,bindings',states',asp')
                    = match_arg (NODE (ans ++ n_ans) id sym args)
                                 binds (record id InRnf states) asp next a_src arity
match_args [ ] bindings states asp next_alt offset arity
 = ([ ],bindings,states,asp)

match_arg :: tree -> bindings -> states -> asp -> label -> a_src -> arity ->
                                                  (abc_assembler,bindings,states,asp)
match_arg (NODE anns id (INTval i) [ ]) binds states asp next a_src arity
 = (code ,binds, states, asp)
   where   code
             = typecheck                                  ++
               [  EqI_a       i a_src                  ]  ++
               escape_false (asp-arity) next
```

```
          typecheck
            = []                                        , if member anns IsINT
            = [  Eq_desc      int_symbolid a_src      ] ++
                 escape_false (asp-arity) next          , otherwise
match_arg (NODE ans id (BOOLval b) [ ]) binds states asp next a_src arity
 = (code ,binds, states, asp)
   where  code
            = typecheck                               ++
              [  EqB_a      b a_src                   ] ++
              escape_false (asp-arity) next
          typecheck
            = []                                        , if member anns IsBOOL
            = [  Eq_desc      bool_symbolid a_src     ] ++
              escape_false (asp-arity) next             , otherwise
match_arg (NODE annots id symbol args) binds states asp next a_src arity
 = (code, bindings", states", asp")
   where  code
                  = [  Eq_desc_arity   (symbol_id symbol) n_arity a_src]   ++
                    escape_false (asp-arity) next                         ++
                    pushargs a_src n_arity                                ++
                    match_sub_args
          bindings' = bind_args args asp' binds
          states'   = record_args args states
          asp'      = asp + n_arity
          n_arity   = # args
          (match_sub_args,bindings",states",asp")
                  = match_args args bindings' states' asp' next asp' arity
```

B.3 Code for the right handside of a rule alternative

```
contractum :: graph -> bindings -> states -> asp -> abc_assembler
contractum (top: node_defs) bindings states asp
 = build_shared_nodes ++ root top bindings' states' asp'
   where  (build_shared_nodes,bindings',states',asp')
             = shared_nodes node_defs (bind top_id 0 bindings) states asp
          NODE annots top_id symbol args    = top

shared_nodes :: [tree] -> bindings -> states ->asp->(abc_assembler,bindings,states,asp)
shared_nodes node_defs bindings states asp
 = (code, bindings", states", asp")
   where  code     = create_cycle_nodes ++ fill_nodes_code
          (create_cycle_nodes,bindings',states',asp')
                   = cycle_nodes node_defs bindings states asp
          (fill_nodes_code,bindings",states",asp")
                   = fill_nodes node_defs bindings' states' asp'
```

```
cycle_nodes :: [tree] -> bindings -> states -> asp -> (abc_assembler,bindings,states,asp)
cycle_nodes [ ] bindings states asp
 = ([ ],bindings,states,asp)
cycle_nodes (NODE annots nodeid sym args:rest) bindings states asp
 = (code, bindings", states", asp")          , if member annots OnACycle
 = cycle_nodes rest bindings states asp     , otherwise
   where  code       = [  Create      ++   code_rest
          bindings'  =  bind nodeid asp' bindings
          states'    =  record nodeid Created states
          asp'       =  asp + 1
          (code_rest,bindings",states",asp") = cycle_nodes rest bindings' states' asp'

fill_nodes :: [tree] -> bindings -> states -> asp -> (abc_assembler,bindings,states,asp)
fill_nodes [ ] bindings states asp
 = ([ ],bindings,states,asp)
fill_nodes (node: rest) bindings states asp
 = (code, bindings", states", asp")
   where  code    =  creation    ++    fill_code    ++    fill_rest_code
          creation
                  =  []                                  , if previously_defined
                  =  [ Create                        ]   , otherwise
          bindings'
                  =  bindings                            , if previously_defined
                  =  bind nodeid asp' bindings           , otherwise
          asp'
                  =  asp                                 , if previously_defined
                  =  asp + 1                             , otherwise
          a_src   =  asp' - bindings' nodeid
          (fill_code,states')    =  fill_node node bindings' states asp' a_src
          (fill_rest_code,bindings",states",asp")  =  fill_nodes rest bindings' states' asp'
          NODE annots nodeid sym args              =  node
          previously_defined                       =  states nodeid = Created

fill_node :: tree -> bindings -> states -> asp -> a_src -> (abc_assembler,states)
fill_node (NODE annots id (INTval i) [ ]) bindings states asp a_src
 = (code, states')
   where  code     =  [   FillI       i a_src                                          ]
          states'  =  record id InRnf states
fill_node (NODE annots id (BOOLval b) [ ]) bindings states asp a_src
 = (code, states')
   where  code     =  [   FillB       b a_src                                          ]
          states'  =  record id InRnf states
fill_node (NODE ans nid (Function sid) args) binds state asp asrc
 = fill_strict (NODE ans nid (Function sid) args) binds state asp asrc, member ans Strict
 = (code, states")                                                    , otherwise
   where  code     =  build_args_code ++ [ Fill sid arity (node_label sid) (asrc+arity)    ]
          (build_args_code,states")  =  build_args args binds states' asp
          states'  =  record nid Unknown state
          arity    =  # args
```

```
fill_node (NODE annots nid (Constructor sid) args) binds states asp asrc
 = (code, states")
   where  code      =  build_args_code ++ [  Fill    sid arity rnf_lab (asrc + arity)          ]
          (build_args_code,states")  =  build_args args binds states' asp
          states'   =  record nid InRnf states
          arity     =  # args

fill_strict :: tree -> bindings -> states -> asp -> a_src -> (abc_assembler,states)
fill_strict (NODE anns nodeid (Function sid) args) bindings states asp 0
 = (code, new_states)
   where  code              =  build_args_code ++   [   Jsr    (reduction_label sid)         ]
          new_states        =  record nodeid InRnf states'
          (build_args_code,states')    =  build_args args bindings states asp
fill_strict (NODE annot id (Function sid) args) bindings states asp asrc
 = (code, states)
   where  code
             =  [   Push_a      asrc                          ]  ++
                fill_code                                        ++
                [   Pop_a   1                                 ]
          (fill_code,states)
             =  fill_strict (NODE annot id (Function sid) args) bindings states (asp+1) 0
fill_strict node bindings states asp a_src
 = fill_node node bindings states asp a_src

build_args :: arg's -> bindings -> states -> asp ->(abc_assembler,states)
build_args [ ] bindings states asp
 = ([ ],states)
build_args (arg:args) bindings states asp
 = (code, states")
   where  code                      =  args_code ++ arg_code
          (args_code,states')       =  build_args args bindings states asp
          (arg_code,states")        =  build_arg arg bindings states' (asp+#args)

build_arg :: arg -> bindings -> states -> asp -> (abc_assembler,states)
build_arg (NodeId annots nodeid) bindings states asp
 = (push ++ evaluation     , new_states   )   , if reduction_needed
 = (push                   , states        )   , otherwise
   where  new_states                =  record nodeid InRnf states
          push                      =  [   Push_a     (asp - bindings nodeid)   ]
          evaluation                =  [   Jsr_eval                             ]
          reduction_needed  =  member annots Strict & ~(states nodeid=InRnf)
build_arg (Term annots node) binds states asp
 = (creation ++ eval_code     , strict_states  )   , if member annots Strict
 = (creation ++ fill_code       , fill_states       )   , otherwise
   where  creation                  =  [   Create                               ]
          (eval_code,strict_states) =  fill_strict node binds states (asp+1) 0
          (fill_code,fill_states)     =  fill_node node binds states (asp+1) 0
```

```
root :: tree -> bindings -> states -> asp -> abc_assembler
root (NODE annots id (INTval i) [ ]) bindings states asp
 = pop_args asp ++
   [  FillI      i 0                                   ,
      Rtn                                              ]
root (NODE annots id (BOOLval b) [ ]) bindings states asp
 = pop_args asp ++
   [  FillB      b 0                                   ,
      Rtn                                              ]
root (NODE annots nid (Constructor sid) args) bindings states asp
 = build_args_code     ++
   [  Fill       sid arity rnf_lab (asp+arity)         ] ++
   pop_args asp        ++
   [  Rtn                                              ]
   where   (build_args_code,states')   =  build_args args bindings states asp
           arity  =  # args
root (NODE annots nid (Function sid) args) bindings states asp
 = build_args_code            ++
   clean_up (# args) asp      ++
   [  Jmp      (reduction_label sid)                   ]
   where   (build_args_code,states')   =  build_args args bindings states asp

redirection :: annots -> node_id -> bindings -> states -> asp -> abc_assembler
redirection annots nodeid bindings states asp
 = [  Fill_a       newroot asp                         ,
      Pop_a        asp                                 ,
      Rtn                                              ]  , if states nodeid = InRnf
 = update_astack newroot (asp-1)   ++
   pop_args (asp-1)                ++
   [  Jsr_eval                                         ,
      Fill_a       0 1                                 ,
      Pop_a        1                                   ,
      Rtn                                              ]  , otherwise
   where   newroot    =  asp - (bindings nodeid)
```

B.4 Auxiliary functions

```
is_strict : : arg -> bool
is_strict (NodeId annots nodeid)  =  member annots Strict
is_strict (Term    annots node   )  =  member annots Strict

get_lhs : : rulealt -> graph
get_lhs (Rewrite     lhs rhs)               =  lhs
get_lhs (Redirect    lhs annots nodeid)  =  lhs

get_rhs : : rulealt -> graph
get_rhs (Rewrite     lhs rhs)               =  rhs
```

```
get_args : : tree -> arg's
get_args (NODE annots nodeid symbol args)   =  args

bring_in_rnf : : annots -> a_src -> abc_assembler
bring_in_rnf annots a_src
 = []                    , if member annots Strict
 = reduce_arg a_src      , otherwise

escape_false :: nr_args -> label -> abc_assembler
escape_false nr_args label
 = [  Jmp_false  label                                 ]  , if nr_args = 0
 = [  Br_true    2                                     ,
      Pop_a      nr_args                               ,
      Jmp        label                                 ]  , otherwise

pop_args : : nr_args -> abc_assembler
pop_args nr_args
 = []                                                     , if nr_args = 0
 = [  Pop_a      nr_args                               ]  , otherwise

update_astack : : a_src -> a_dst -> abc_assembler
update_astack a_src a_dst
 = []                                                     , if a_src = a_dst
 = [  Update_a  a_src a_dst                            ]  , otherwise

pushargs : : a_src -> arity -> nr_args -> abc_assembler
pushargs a_src arity nr_args
 = []                                                     , if nr_args = 0
 = [  Push_args    a_src arity nr_args ]                  , otherwise

bind_args : : arg's -> offset -> bindings -> bindings
bind_args (NodeId annots nodeid:rest) n old
 = bind_args rest (n-1) (bind nodeid n old)
bind_args (Term annots (NODE annots' nodeid sym args):rest) n old
 = bind_args rest (n-1) old                               , if nodeid = ""
 = bind_args rest (n-1) (bind nodeid n old)               , otherwise
bind_args [ ] n bindings
 = bindings

bind : : node_id -> offset -> bindings -> bindings
bind nodeid offset bindings name
 = offset                                                 , if name = nodeid
 = bindings name                                          , otherwise

empty_bindings : : bindings
empty_bindings nodeid  =  error ("NodeId not bound in this stack frame: "++nodeid)

record_args : : arg's -> states -> states
record_args (NodeId annots id:rest) states
 = record_args rest (record id InRnf states)        , if member annots Strict
 = record_args rest (record id Unknown states)      , otherwise
```

```
record_args (Term annots (NODE n_annots id sym args):rest) states
 = record_args rest (record id InRnf states)          , if member (annots++n_annots) Strict
 = record_args rest (record id Unknown states)        , otherwise
record_args [ ] states
 = states

record : : node_id -> status -> states -> states
record nodeid status states name
 = states name                                        , if nodeid = ""        || no node-id
 = status                                             , if name = nodeid
 = states name                                        , otherwise

initial_status : : states
initial_status name =  Undefined

clean_up : : num -> num -> abc_assembler
clean_up keep  0     = [ ]
clean_up 0     drop = [ Pop_a      drop                       ]
clean_up keep  drop = [ Update_a  (keep-1) (keep+drop-1)] ++ clean_up (keep-1) drop

node_label : : symbolid -> label
node_label symbolid  =  "n_" ++ symbolid

apply_label : : symbolid -> label
apply_label symbolid  =  "a_" ++ symbolid

alt_label : : symbolid -> num -> label
alt_label symbolid n    =  symbolid ++ show n

reduction_label : : symbolid -> label
reduction_label symbolid    =  alt_label symbolid 1

cycle_lab : : label
cycle_lab   =  "_cycle"

rnf_lab : : label
rnf_lab       =  "_rnf"

type_error : : label
type_error  =  "type_error"

symbol_id : : symbol -> symbolid
symbol_id (Symbol id)          =  id
symbol_id (Function id)        =  id
symbol_id (Constructor id)   =  id
symbol_id symbol                 =  error ("No id in the symbol: " ++ show symbol)

int_symbolid : : symbolid
int_symbolid       =  "INT"

bool_symbolid : : symbolid
bool_symbolid    =  "BOOL"
```

Appendix C
Artificial Neural Networks

This appendix contains a complete definition of a set of tools to describe and investigate the behaviour artificial neural networks. These tools are used to describe a number of network models in chapter 6. Only the specification of the network for the N-queens problem is not covered completely in the text, so the complete definition is shown here.

C.1 Network description tools

```
weights_net     ==  [weights_layer]  || the synaps weights in a network
weights_layer   ==  [weights]        || the synaps weights in a layer
weights         ==  [weight]         || the weights defining one synapse
weight          ==  num
input_state     ==  [input_value]    || list of simultaneous input_values
input_value     ==  value
output_state    ==  [output_value]   || list of simultaneous output_values
output_value    ==  value
net_state       ==  [state]          || the states of all layers in a net
state           ==  [value]          || the states of neurons in a layer
target_state    ==  [target_value]   || the target output_state
target_value    ==  value
potential       ==  num              || the internal potential of a neuron
value           ==  num              || used binary or continuously by application
vector          ==  [num]
vectors         ==  [vector]
vector_id       ==  num
class           ==  [num]            || list to be compatible with output_state
target_class    ==  class            || used in learning vector quantisation
```

```
synapse        ==  weights -> input_state -> potential
generator      ==  potential -> output_value
element        ==  weights -> neuron
neuron         ==  input_state -> output_value
```

Functions determining the evaluation order

```
evaluate the state of of a layered network; state!0 is the input state
feed_forward :: element -> weights_net -> input_state -> net_state
feed_forward element weights input
 = state
   where  state
               = (input ++ constant_neurons):
                 [eval_layer element (weights!l) (state!l) ++constant_neurons
                 | l <- [0..#weights -1]]

constant_neurons :: state
constant_neurons =   rep nr_constant_neurons high

feed_back_syn :: element -> weights_net -> input_state -> [output_state]
feed_back_syn element weights input
 = to_limit(iterate(select_output.feed_forward element weights) input)

feed_back_asyn :: element -> weights_net -> input_state ->[output_state]
feed_back_asyn element [weights] input
 = to_limit (iterate (eval_one_by_one element weights) input)

annealing :: num -> element -> weights_net ->input_state->[output_state]
annealing t element [weights] input
 = to_limit (disturb t (eval_one_by_one element weights) input)

disturb :: num -> (state->state) -> state -> [state]
disturb t eval state
 = state : disturb (annealing_rate*t) eval new_state
   where  v              = sum state
          n              = # state
          new_state  =  eval [state!i + noise t (n*i+v) | i <- [0..n-1]]
          annealing_rate  =  0.8

noise :: num -> num -> num
noise t s =  t*(1 - (random (23459*entier s) mod 101)/50) || pseudo random within [-t,t]

eval_layer :: element -> weights_layer -> input_state -> output_state
eval_layer element layer input =  [element weights input | weights <- layer]

select_output :: net_state -> output_state
select_output state
 = output_state                                              , if nr_constant_neurons = 0
 = take (#output_state-nr_constant_neurons) output_state  , otherwise
   where output_state =  last state
```

```
eval_one_by_one :: element -> weights_layer -> input_state -> output_state
eval_one_by_one element layer input  =  eval_neurons element input layer [0..# layer -1]
   || evaluate the neurons in a feed back network one after another in pseudo random order

eval_neurons :: element -> state -> weights_layer -> [num] -> state
eval_neurons element state w_layer [ ] =  state ++ constant_neurons
eval_neurons element state w_layer indices
 = eval_neurons element new_state w_layer rest
   where  n          =  random (1993+entier (sum state)) mod # indices
          index      =  indices ! n
          rest       =  remove n indices
          new_state  =  update index new_value state
          new_value  =  element (w_layer!index) state
          || Update one neuron at each iteration; taken pseudo random of the list of indices

random :: num -> num
random seed =  (16807*seed) mod 2147483647      || 2147483647=2^31-1

learn :: weights_net -> [input_state] -> [target_state] -> weights_net
learn weights (sample:samples) (target:targets)
 = learn (adapt_weights weights sample target) samples targets
learn weights [ ] [ ]
 = weights
```

Functions determining the specific network

```
high                  :: value                      || The upper limit of the values
low                   :: value                      || The lower limit of the values
nr_constant_neurons   :: num                        || The number of constant neurons
compute_state         :: weights_net -> input_state -> [state]
compute_output        :: weights_net -> input_state -> output_state
adapt_weights         :: weights_net -> input_state -> target_state -> weights_net
gain                  :: num                        || Determines the learing rate
a                     :: num                        || Determines the size of the initial weights
initial_net           :: [num] -> weights_net
show_elem             :: value -> char
show_net              :: weights_net -> [char]
```

The synaptic functions

```
wsum :: synapse
wsum   =  inner_product

inner_product :: synapse
inner_product v1 v2    =  sum (map2 (*) v1 v2)   || or sum [v1!i * v2!i | i <- [0..# v1 -1] ]

neocognitron_synapse :: synapse
neocognitron_synapse weights input
 = (1+excitation)/(1+inhibition) - 1
   where  excitation =    sum[weights!i*input!i|i<-[0..#weights-1];weights!i>0]
          inhibition =   -sum[weights!i*input!i|i<-[0..#weights-1];weights!i<0]
```

The generator functions

```
hard_limit :: generator
hard_limit v
 = high  , if v > threshold
 = low   , otherwise

sgn :: generator
sgn v
 =  1    , if v > 0
 = -1    , otherwise

sigmoid :: generator
sigmoid v  =  (high-low) / (1+exp (-sharpness*v)) + low

sigmoid' :: generator          || the derivate of sigmoid
sigmoid' v  =  ((high-low)*sharpness*exp (-sharpness*v))/((1+exp (-sharpness*v))^2)

sharpness :: num               || determines the shape of the sigmoid
sharpness =  2

clip :: generator
clip v
 = low   , if v <= low
 = v     , if low < v <= high
 = high  , if high < v
```

Some well-know elements

```
perceptron_b :: element
perceptron_b weights    =  sgn        . wsum weights    || For binary values

perceptron_c :: element
perceptron_c weights    =  sigmoid    . wsum weights    || For continuous values

hopfield_b :: element
hopfield_b weights      =  hard_limit . wsum weights    || For binary values

hopfield_c :: element
hopfield_c weights      =  clip       .  wsum weights   || For continuous values
```

Functions to trace the evolution of the state and the learning of a net

```
evolution :: weights_net -> [input_state] -> [target_state] -> [input_state] -> [char]
evolution initial_weights samples targets inputs
 = lay [ lay [ show_state state | state <- compute_state weights input] | input <- inputs ] ++
   show_w_net weights
   where weights   =  learn initial_weights samples targets
```

```
|| trace take 4 arguments:
||  - f       : the function to be learned
||  - size    : the size of the network to be generate
||  - m       : the number of learning cycles between status reports
||  - n       : the total number of learning cycles to preform

trace :: (state -> state) -> [num] -> num -> num -> [char]
trace f size m n
 = "gain: " ++ show gain ++ " a: " ++ show a ++ "\n" ++
   concat [eval_perf (weights!i) (i*m) all_samples all_targets |i<-[0..reps]] ++
   show_net (weights!reps)
   where  weights      = initial_net size: [learn (weights!i) (inputs!i) (targets!i) |i<-[0..]]
          inputs       = chop m input_seq
          targets      = chop m [f input | input <- input_seq]
          input_seq    = all_samples ++ random_patterns dimension
          all_samples  = patterns dimension
          all_targets  = [f input | input <- all_samples]
          reps         = n div m
          dimension    = size!0

eval_perf :: weights_net -> num -> [state] -> [state] -> [char]
eval_perf weights n samples targets
 = "score after " ++ show n ++ " learning cycles: " ++
   showfloat 0 (100 * score) ++ " %\n" ++
   show_sot samples outputs targets
   where  score   = sum [1|i<-[0..#samples-1];outputs!i=targets!i]/(# samples)
          outputs = [compute_output weights input | input <- samples]

show_sot :: [input_state] -> [output_state] -> [target_state] -> [char]
show_sot samples outputs targets
 = lay   ["input:\t"  ++ show_state (samples!i) ++
          "\toutput:\t" ++ show_state (outputs!i) ++
          "\ttarget:\t" ++ show_state (targets!i) | i <- [0..#samples-1]]

show_state :: state -> [char]
show_state state    = "|" ++ [show_elem elem | elem <- state] ++ "|"

show_w_net :: weights_net -> [char]
show_w_net net      = layn [show_w_layer layer | layer <- net]

show_w_layer :: weights_layer -> [char]
show_w_layer layer =     lay [show_w_vector vector | vector <- layer]

show_w_vector :: weights -> [char]
show_w_vector vector =  concat ["\t" ++ showfloat 2 elem| elem <- vector]
```

Some list manipulation functions

```
update :: num -> * -> [*] -> [*]
update 0 new (a:x)  =  new:x
update n new (a:x)  =  a: update (n-1) new x
```

```
remove :: num -> [*] -> [*]
remove 0 (a:x)       =  x
remove n (a:x)       =  a : remove (n-1) x

to_limit::[*]->[*]
to_limit (a:b:x)
 = [a]                   , if a=b
 = a: to_limit (b:x)     , otherwise

scan_list :: num -> * -> [*] -> num
scan_list i x (y:rest)
 = i                          , if x =   y
 = scan_list (i+1) x rest     , otherwise

chop :: num -> [*] -> [[*]]
chop n seq =  take n seq : chop n (drop n seq)

normalise :: vector -> vector
normalise vector
 = [element / l | element <- vector]      , if l > 0
 = error "cannot normalise 0-vector"  , otherwise
   where l =   length vector

length :: vector -> num
length vector  =  sqrt (sum [element^2 | element <- vector])

patterns :: num -> [state]                          || Generates all binary patterns of length n
patterns n
 = pats n [[ ]]
   where   pats 0 states   =  states
           pats n states
           =  pats (n-1) ([low : state | state <- states] ++ [high: state | state <- states] )

random_patterns :: num -> [state]                   || Generates a list of random states
random_patterns dimension
 = f 3
   where   f n      =  pats!(n mod limit):f (random n)  || Pseudo random
           pats  =  patterns dimension
           limit  =  2^dimension

random_input :: num -> [state]
random_input dimension =  chop dimension (random_values dimension)

random_values :: num -> [value]
random_values n
 = value : random_values (random n)
   where   value
              =  high  , if n mod 2 =  1
              =  low    , otherwise
```

Some functions to be learned

```
xor_fun :: input_state -> output_state
xor_fun inputs
 = [high]    , if # (filter (=high) inputs) = 1
 = [low]     , otherwise

and_fun :: input_state -> output_state
and_fun inputs
 = [high]    , if # (filter (=low) inputs) = 0
 = [low]     , otherwise

or_fun :: input_state -> output_state
or_fun inputs
 = [high]    , if # (filter (=high) inputs) >= 1
 = [low]     , otherwise

majority_fun :: input_state -> output_state
majority_fun inputs
 = [high]    , if # (filter (=high) inputs) > # inputs/2
 = [low]     , otherwise
```

C.2 A feedback network to solve the N-queens problem

```
high    =  1.0
low     =  0.0
queen   =  high
empty   =  low
nr_constant_neurons =  0

queen_generator v              =  clip (v + 1/3)
queen_element weights          =  queen_generator.wsum weights
compute_state                  =  feed_back_asyn queen_element
compute_output weights input   =  select_output (compute_state weights input)

show_net =  show_w_net
show_elem value
 = 'Q'    , if v > 0.9
 = 'q'    , if v > 0.7
 = '*'    , if v > 0.5
 = '+'    , if v > 0.3
 = '.'    , if v > 0.1
 = ' '    , otherwise
   where v =  (value - low) / (high - low)

initial_net [n]  =  [[[matrix_value i j n | i <- [0..(n^2-1)]]  | j <- [0..(n^2-1)]]]
```

```
matrix_value p1 p2 n
 = 1      , if p1 = p2
 = -1/2   , if i=k ∨ j=l ∨ abs (i-k) = abs (j-l)
 = 0      , otherwise
   where  (i,j)  =  coordinates p1 n
          (k,l)  =  coordinates p2 n

coordinates :: num -> num -> (num,num)
coordinates field_number board_size
 = (row,col)
   where  row   =  field_number div board_size
          col   =  field_number mod board_size

empty_board :: num -> input_state
empty_board n  =  rep (n^2) empty

test :: num -> [num] -> [char]
test n positions
 = monitor (compute_state (initial_net [n])  (place_queens positions (empty_board n)))

place_queens [ ]        board   =  board
place_queens (p:rest) board   =  place_queens rest (update p queen board)

monitor :: [output_state] -> [char]
monitor outputs  =  concat [show_board board | board <- outputs]

show_board :: state -> [char]
show_board state
 = border++lay [show_state row | row <- take n (chop n state)]++border
   where  border  =  "+" ++ rep n '-' ++ "+\n"
          n       =  entier (sqrt (# state))
```

References

Aho, A.V., Sethi, R., Ullman, J.D. (1986). *Compilers - Principles, Techniques and Tools*. Addison-Wesley series in Computer Science.

Amit, D.J., Gutfreund, H., Sompolinsky, H. (1985). Storing infinite numbers of patterns in a spin-glass model of neural networks. *Physical Review Letters*, **55**, pp 1530-1533.

Amit, D.J., Gutfreund, H., Sompolinsky, H. (1985). Spin-glass models of neural networks. *Physical Review A*. **32**(2), pp 1007-1018.

Andres, V. (1990). The N queens problem: A neural Algorithm. Proceedings iASTED Int. Conf. on Modelling, Simulation and Optimization. May 22-24, 1990. Montreal, Canada.

Backus, J.W. (1987). Can programming be liberated from the van Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, **21**, pp 613-641.

Bakel, S.J. van, (1990) *A First Attempt in Polymorphic Type Assignment using Intersection Types*. Techincal Report 90-9, May 1990, Department of Computer Science, University of Nijmegen, the Netherlands.

Barendregt, H.P. (1984) *The lambda-calculus, Its Syntax and Semantixs* (revised edition). Studies in logic and foundations of mathematics. **103**, North-Holland, Amsterdam.

Barendregt, H.P., Eekelen, M.C.J.D. van, Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., Sleep, M.R. (1987a). Term Graph Rewriting. Proceedings of Parallel Architectures and Languages Europe (PARLE), part II, Eindhoven, The Netherlands. *Springer Lec. Notes Comp. Sci.* 259. pp 141-158.

Barendregt, H.P., Eekelen, M.C.J.D. van, Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., Sleep, M.R. (1987b). Towards an Intermediate Language based on Graph Rewriting. Proceedings of Parallel Architectures and Languages Europe (PARLE), part II, Eindhoven, The Netherlands. *Springer Lecture Notes on Computer Science*. **259**. pp 159-175.

Barendregt, H.P., Eekelen, M.C.J.D. van, Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., Sleep, M.R. (1988). Towards an Intermediate Language based on Graph Rewriting. Revised version. *Journal of Parallel Computing* **9** with selected papers of the conference on Parallel Architectures and Languages Europe (PARLE), Eindhoven, The Netherlands. North-Holland pp 163-177.

Beizer, B. (1983) Software Testing Techniques. Van Nostrand Reinhold electrical/computer science and engineering series. Van Nostrand Reinhold Company. New York. ISBN: 0-442-24592-0.

Bird, R. Wadler, P, (1988) *Introduction to functional programming*. Prentice Hall, New York.

Boute, R.T. (1988). System Semantics: Principles, Applications, and Implementation. ACM Transactions on Programming Languages and Systems, Vol. **10**. No. 1. January 1988, pp 118-155.

Brus, T. Eekelen, M.C.J.D. van, Leer. M. van, Plasmeijer, M.J. (1987). Clean - A Language for Functional Graph Rewriting. Proc. of the Third International Conference for Functional Programming Languages and Computer Architecturs (FPCA '87), Potland, Oregon, USA. *Springer Lecture Notes on Computer Science* **274**, pp 364-384.

Burn, G.L., Peyton Jones, S.L., Robson, J.D. (1988). The spineless G-machine. Proceedings of the 1988 ACM Conference on LISP and Functional Programming.

Burstall, R.M., MacQueen, D.B., Sanella, D.T. (1980) Hope: An experimental Applicative Language. Proceedings of the 1980 LISP Conference. pp 136-143.

Church, A. (1932 and 1933). A set of Postulates for the Foundation of logic. *Annals. of Math.* (2) **33** pp 346-366 and **34** pp 839-864.

Coolen, A.C.C. (1989). *Inleiding Neurale Netwerken*. Vakgroep Medische en Fysiologische Fysika, Rijksuniversiteit Uterecht.

Curry, H. B., Feys, R. (1958). *Combinatory Logic*. **1**, North Holland.

DeMarco, T. (1987). *Structured analysis and system specification.* Yordon Press, New York.

Eekelen, M. C. J. D. van, Plasmeijer, M.J. (1986), Specification of rewriting strategies in Term Rewriting Systems. Proceedings of the workshop on Graph Reduction, Santa Fe, New Mexico. *Springer Lec. Notes Comp. Sci.* **279**, 215-239.

Eekelen, M. C. J. D. van. (1988) *Parallel Graph Rewrting, Some Contributions to its Theory, its Implementation and its Application.* Ph. D. Thesis, University of Nijmegen.

Eekelen, M. C. J. D. van, Nöcker E. G. J. M. H., Plasmeijer, M. J. ,Smetsers, J. E. W. (1989) *Concurrent Clean.* Technical report 89-18, October 1989. University of Nijmegen. The Netherlands.

Ellacott, S.W. (1990) An analysis of the delta rule. Proceedings INNC (International Neural Network Conference), Paris, 1990. Volume II. Kluwer Academic Publisers. pp 956-990.

Fehr, E. (1989), *Semantik von Programmiersprachen.* Springer-Verlag. Berlin.

Field, A.J., Harrison, P.G. (1988), *Functional programming.* Addison-Wesley.

Gielen, C.C.A.M.(1988) *Lecture notes caputcollege Neurophysics*, Department of Medical and Biophysics, Nijmegen University. The Netherlands.

Gordon, M.J.C. (1979), *The denotational description of programming languages.* Springer-Verlag, New York.

Groningen. J.C. van. (1990). M. Sc. thesis. University of Nijmegen. To appear.

Hebb, D. O. (1949), *The Organization of Behaviour.* Wiley, New York.

Hecht-Nielsen, R. (1987a). Counter-propagation networks. *IEEE First International Conference on Neural Networks*, Vol II, pp 19-32.

Hecht-Nielsen, R. (1987b). Neurocomputer application. *Proc. National Computer Conf. AFIPS.* pp 239-244, 1987.

Hecht-Nielsen, R. (1988). Neurocomputing: picking the neural brain, *IEEE Spectrum*, Vol **25**, no 3 pp 36-41, 1988.

Hoare, C.A.R. (1962). Quicksort. *Computer Journal*, **5**, Nr 1, pp 10-15.

Hobbie, R.K. (1978). *Intermediate Physics for Medicine and biology.* Wiley New York.

Hopfield, J.J. (1982) Neural Networks and Physical systems with Emergent Collective Computional Abilities. *Proc. Natl. Acad. Sci. USA*, **79**, pp 2554-2558.

Hopfield, J.J. and Tank, D.W. (1985). "Neural" Computation of Decisions in Optimization Problems. *Biologocal Cybernetics*, **52**, pp 141-152.

Hudgkin, A.L. and Huxley A.F. (1952). "A quantitative description of membrane current and its application to conduction and excitation in nerve" *Journal Physiology*, **117**: 500-544.

Hughes, J. (1989). Why Functional Programming matters. *The computer journal*, Vol **32**, no 2, 1989 pp 98 - 107.

INNC 90. Proceedings INNC (International Neural Network Conference), Paris, 1990. Volume I & II. Kluwer Academic Publisers. Dordrecht

Johnson, Th. (1984). Efficient compilation of lazy evaluation. Proceedings of the ACM SIGPLAN '84, Symposium on Compiler Construction. *SIGPLAN Notices* 19/6.

Johnson, Th. (1987). *Compiling Lazy Functional Programming languages.* Dissertation at Chalmers University, Götenborg, Sweden. ISBN 91-7032-280-5.

Joosten, S.M.M. (1989). The use of functional programming languages in software development. Dissertation Enschede. ISBN: 90-902729-7.

Kernighan, B.W., Ritchiw, D.M. (1978). *The C programming language.* Prentice-hall software series.

Kirkpatrick, S., Gelatt, C.D., Vecchi, M,P. (1983) Optimization by Simulated Annealing. *Science*, **220**, pp 671-680.

Klop, J.W. (1987), *Term Rewriting systems: a tutorial.* Center for Mathematics and Computer Science, CWI Amsterdam, The Netherlands. Note CS-N8701.

Kohonen, T. (1977), *Associative Memory-A System Theoretic Approach.* Springer, New York.

Kohonen, T. (1988a), *Self-Organisation and Associative Memory.* Springer, New York.

Kohonen, T. (1988b), An Introduction to Neural Computing, *Neural Networks*, **1**, pp 3-16.

Kohonen, T. (1988c), *Neural Networks*, **1**, pp 17-61.

Koopman, P.C. (1960). *Neurotransmitters and their chemical derivates. A study on the relationship between structure and activity*. Dissertation. R.C. University Nijmegen, The Netherlands.

Koopman, P.W.M. (1987). Interactive Programs in a Functional Language: A Functional Implementation of an Editor. *Software-Practice and Experience*, **17**(9), pp 609-622.

Koopman, P.W.M., Nöcker, E.G.J.M.H.(1988) *Compiling Functional Languages to Term Graph Rewrite Systems*. Internal Report no. 88-1, Department of Computer Science, University of Nijmegen, The Netherlands.

Koopman, P.W.M., Eekelen, M.C.J.D. van, Nöcker, E.G.J.M.H., Smetsers, J.E.W., Plasmeijer, M.J. (1990). *The ABC-machine: A Sequential Stack-based Abstract Machine For Graph Rewriting*. Internal Report, University of Nijmegen, to appear in 1990.

Koopman. P.W.M., Rutten, L.M.W.J., Eekelen, van M.C.J.D. and Plasmeijer, M.J. (1990), *Functional Description of Neural Networks*, Proceedings INNC (International Neural Network Conference), Paris, 1990. Volume II. Kluwer Academic Publisers. pp 701-704.

Landin, P.J. (1964). The mechanical evaluation of expressions. *Computer Journal*. **6**, pp. 308 - 320.

Ligner, G.T. (1975) A Mathematical Approach to Language Design. *Proceedings of the second ACM symposium on priciples of programming languages*, Palo alto.

Lippman, R.P. (1987). An Introduction to Computing with Neural Nets. IEEE ASSP Magazine April 1987. pp 4-22

Milner, R.A. (1978). Theory of Type Polymorphism in Programming. *Journal of Computer System Sciences*, **17**, no 3. pp 348 - 375.

Minsky, M.L., Papert, S.A. (1969), *Perceptrons*. MIT, Cambridge (Mass).

Mycroft, A. (1984). Polymorphic type schemes and recursive definitions. Proc. of the 6th Int Conf on Programming. *Springer Lecture Notes on Computer Science* **167**. pp 217 - 228.

Nöcker, E.G.J.M.H. (1989) *The PABC Simulator v0.5. Implemenation Manual*. Technical report no. 89-19, October 1989. Department of Informatics. University of Nijmegen. The Netherlands.

Nöcker, E.G.J.M.H. (1990) *Strictness analysis and pattern matching*. Technical report. University of Nijmegen. The Netherlands. To appear.

Peyton Jones, S.L. (1987). *The implementation of functional programming languages*. Prentice-Hall. ISBN 0-13-453325-9.

Plasmeijer, M.J., Eekelen, van M.C.J.D.(1991). *Functional Programming and Parallel Graph Rewritting*. To appear by Eddison Wesly. Currently available as preprint at the department of computer science, university of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands.

Reif, F. (1965). *Fundamentals of statistical and thermal physics*.McGraw-Hill

Reichl, L.E. (1980), *A Modern Course in Statistical Physics*. E.A. London.

Ridgway, W.C. (1962). *An adaptive Logic System with Generalizing Properties*, PhD thesis, Standford Electronnics Labs, Stanford University, April 1962.

Rosenblatt,.F. (1958). The perceptron: A probalistic model for information storage and organization in the brain. *Psychoanlytical Review*, **65**, pp 386-408.

Rosenblatt, F. (1962), *Principles of Perceptrons*. Spartan, Washington.

Rumelhart, D.E., Hinton, G.E., Williams, R.J. (1986). Learning internal representations by error propagation. In Rumelhart & McClelland: *Parallel Distributed Processing: Explorations in the microstructure of Cognition. Vol 1: Foundations*. MIT Press.

Schadé. J.P. (1984). *Onze hersenen*, Aula pocket 721, Het Spectrum, Utrecht/Antwerpen.

Schönfinkel, M. (1924). Über die Bausteine der mathematischen Logik. Math. Annalen **92**. pp 305-316.

Simpson, P.K. (1990) *Artificial Neural Systems*. Pergamon Press. ISBN 0-08-037894-3

Smetsers, J.E.W. (1987). *Parallel graph reduction on a distributed machine*. M. Sc. thesis, Philips Research Laboratories, University of Nijmegen, October 1987.

Smetsers, J.E.W. (1989). *Compiling Clean to Abstract ABC-Machine Code*. Technical Report 89-20. Department of Computer Science, University of Nijmegen, the Netherlands.

Smetsers, J.E.W. Nöcker, E.G.J.M.H., Koopman, P.W.M., Plasmeijer, M.J., Eekelen, van M.C.J.D.(1990). *The parallel ABC-machine.* To appear.

Stoy J.E. (1977). *Denotational Semantics: The Scott-Strachy Approach to Programming Language Theory.* Cambridge (Massachusetts), MIT Press.

Tanenbaum, A.S. (1984). *Structured computer organization*, second edition. Prentice-Hall.

Tank, D.W., Hopfield, J.J. Collective Computation in Neuronlike Circuits. *Scientific American.*

Turner, D. A., (1976). *SASL Language Manual*; (1979) *SASL Language Manual, "combinators" version.* University of St Andrews.

Turner, D. A., (1979). A New Implementation Technique for Applicative Languages. *Software-Practice & Experience*, **9**(1), pp 31-49.

Turner, D. A., (1985). Miranda: A non-strict functional language with polymorphic types. Proc. of the conference on Functional Programming Languages and Computer Architecture, *Springer Lecture Notes on Computer Science* **201**. pp 1 - 16.

Weijers, G.A. (1990), *ABC-machine implementation Guide*, University of Nijmegen. To appear.

Widrow, B. (1962). Generalization and information storage in networks of Adaline 'neurons', in *self-organzing systems 1962*, Yovits, M.C., Jacobi, G.T. and Goldstien, G. eds, Spartan Books, Washington, DC, pp 435-461.

Widrow, B. and Streams, S.D.(1985), *Adaptive signal processing*, Prentice hall, Englewood Cliffs, N,J.

Widrow, B.,Winter, R. (1988) Neural nets for adaptive filtering and adaptive pattern recognition, *IEEE Computer*, March 1988.

Wirth, N. (1971) The programming language PASCAL. Acta Informatica 1, 1971. pp 35-63.

Index

Keywords and function names

Samenvatting

Functionele programma's als uitvoerbare specificaties

Dit proefschrift behandelt het gebruik van functionele programmeertalen voor het opstellen van specificaties die door een computer kunnen worden uitgevoerd.

Een specificatie is een zeer nauwkeurige beschrijving in een geschikte beschrijvingstaal. Een bruikbare specificatie is duidelijk, foutloos en voldoende compleet. De gebruikte beschrijvingstaal mag geen dubbelzinnigheden bevatten en moet krachtig genoeg zijn om compacte specificaties mogelijk te maken. In de informatica worden specificaties gebruikt om eigenschappen en gedragingen van machines en programma's vast te leggen. Een specificatie in een functionele programmeertaal is niet alleen een beschrijving van de eigenschappen, maar geeft ook een algoritme: een voorschrift om die eigenschappen te verwezenlijken.

In traditionele programmeertalen bestaan programma's uit een lange reeks bevelen. Deze talen heten daarom imperatieve talen. Het gevolg van een bevel is afhankelijk van de eerder uitgevoerde bevelen. Dergelijke programma's zijn daardoor vaak moeilijk te begrijpen en het is niet eenvoudig hun correctheid aan te tonen.

In een functionele taal bestaat een programma uit een reeks functiedefinities en een beschrijving van de gewenste oplossing in termen van deze functies. De functies geven eigenschappen van het beschreven object. Zij worden gebruikt om de beschrijving van het resultaat in de meest eenvoudige vorm te krijgen. De beschrijving moet zo gekozen worden dat niet alleen de goede oplossing beschreven wordt, maar ook een manier om die oplossing te bereiken. Doordat tijdens het vereenvoudigen de betekenis van de beschrijving behouden blijft zijn programma's in een functionele taal duidelijk en is de correctheid relatief eenvoudig te bewijzen. Dergelijke programma's zijn bovendien vaak erg kort en bondig.

Om vertrouwen te krijgen in de correctheid van de beschrijving en de geschiktheid van het beschreven produkt wordt vaak een prototype gemaakt. Een prototype is een gedeeltelijke, relatief snel te maken implementatie van het beschreven produkt. Het prototype geeft inzicht in het gedrag van het beschre-

ven produkt, maar heeft een aantal beperkingen ten opzichte van het definitieve produkt. Bij een prototype van een programma gaat het primair om een bepaald gedrag en bijvoorbeeld niet om de snelheid of de manier waarop het programma wordt aangeroepen.

Het is aantrekkelijk programmeertalen als beschrijvingstaal te gebruiken. Programmeertalen zijn doorgaans zo goed gedefiniëerd dat het duidelijk is wat de beschrijving precies betekent. De implementatie van de programmeertaal kan gebruikt worden om een aantal aspecten van de beschrijving automatisch te controleren. Een compiler controleert bijvoorbeeld of er geen taalfouten gemaakt zijn en of alle gebruikte termen gedefiniëerd zijn. Bovendien is de specificatie een programma dat door een computer kan worden uitgevoerd, de beschrijving is dus zijn eigen prototype.

Ondanks deze voordelen worden programmeertalen zelden als beschrijvingstaal gebruikt: zij zijn meestal niet krachtig genoeg om korte en duidelijke beschrijvingen mogelijk te maken. Wanneer we toch zo'n programmeertaal zouden gebruiken dan wordt de beschrijving zó lang dat de duidelijkheid verloren gaat. Bovendien verdwijnt op die manier het verschil tussen beschrijving en produkt.

Functionele programmeertalen zijn hoog-niveau-programmeertalen met een grote uitdrukkingskracht, ze lijken daarom geschikter als beschrijvingstaal dan de traditionele imperatieve talen. In dit proefschrift worden enkele beschrijvingsmethoden geïntroduceerd om de geschiktheid van functionele programmeertalen als beschrijvingstaal te onderzoeken. Aan de hand van enige voorbeelden worden deze beschrijvingen vergeleken met traditionele beschrijvingen. De bruikbaarheid van deze beschrijvingsmethoden voor uitgebreide specificaties wordt onderzocht door ze toe te passen op enkele voorbeelden uit het dagelijks leven van een informaticus.

In hoofdstuk 2 wordt een machine-beschrijvingsmethode geïntroduceerd die bestaat uit meerdere lagen. In de onderste laag worden de onderdelen van de machine, bijvoorbeeld de geheugens, beschreven. De volgende laag beschrijft de instructies van de machine door het effect van iedere instructie op de onderdelen van de machine aan te geven. De verkregen beschrijving wordt vergeleken met een traditionele beschrijving. De machine-beschrijving volgens de nieuwe methode blijkt iets langer te zijn dan de oude specificatie. Dit wordt veroorzaakt door de overzichtelijke manier van noteren die we onszelf hebben opgelegd. De functionele beschrijving blijkt erg duidelijk te zijn en is bovendien iets directer dan de traditionele beschrijving. Het is tevens een bruikbaar prototype.

Graafherschrijfsystemen zijn een zeer geschikt berekeningsmodel voor de implementatie van functionele programmeertalen. Een programma in een functionele programmeertaal worden eerst vertaald naar een equivalent programma in de graafherschrijftaal Clean. Een Clean-programma wordt vervolgens vertaald naar een programma voor een abstracte, imperatieve graafherschrijfmachine: de ABC-machine. Deze abstracte machine is ontworpen om aan te kunnen geven hoe een Clean-programma wordt uitgevoerd op imperatieve machines zonder last te hebben van de beperkingen van een concrete machine.

Als laatste stap wordt een ABC-programma vertaald naar code voor een concrete machine. Het blijkt dat via deze tussenstappen uitstekende implementaties van functionele programmeertalen verkregen worden. De beschrijvingsmethode uit hoofdstuk 2 wordt in hoofdstuk 3 gebruikt voor de beschrijving van de ABC-machine. De gegeven specificatie is de enige machine-beschrijving die beschikbaar is en voldoet goed in het gebruik.

In hoofdstuk 4 wordt de geschiktheid van functionele talen voor het beschrijven van vertalingen van programmeertalen onderzocht. De te transformeren taal wordt gerepresenteerd door een datastructuur in de functionele beschrijvingstaal. De vertaling is een functie, die de gewenste transformatie op de datastructuren uitvoert. Opnieuw voldoet de specificatie in een functionele programmeertaal goed in vergelijking met andere beschrijvingen.

De vertaling van de graafherschrijftaal Clean naar instructies voor de in hoofdstuk 3 beschreven ABC-machine wordt in hoofdstuk 5 behandeld. Deze beschrijving dient als ontwerp voor de echte vertaler. Het uiteindelijke produkt bevat veel extra optimalisaties. Het is mogelijk om al die optimalisaties in de specificatie op te nemen, maar dan zou het basisschema van de vertaling onvoldoende duidelijk naar voren komen. De beschrijvingen uit hoofdstuk 3 en 5 zijn samen gebruikt om Clean programma's uit te voeren en om hun gedrag te bestuderen voordat de beschreven producten gemaakt werden.

In hoofdstuk 6 worden een aantal kunstmatige neurale netwerken beschreven in een functionele taal. Deze beschrijvingen zijn een zinvolle aanvulling op de traditionele wiskundige beschrijvingen. Wiskundige specificaties zijn, vaak niet complete, beschrijvingen van eigenschappen van deze neurale netwerken, maar geven niet aan hoe dit gedrag op een efficiënte manier bereikt kan worden. De ontwikkelde beschrijvingen laten de overeenkomsten en verschillen tussen de netwerken duidelijk zien en zijn gebruikt als prototype om een aantal niet wiskundig te bepalen eigenschappen te onderzoeken.

Als conclusie kunnen we stellen dat een functionele programmeertaal met succes is gebruikt voor een aantal uitgebreide beschrijvingen die de vergelijking met andere specificaties glansrijk kunnen doorstaan.

Voor alle in dit proefschrift gegeven beschrijvingen is de functionele programmeertaal Miranda gebruikt. Deze taal is niet speciaal als beschrijvingstaal ontworpen, maar is desalniettemin goed bruikbaar als beschrijvingstaal. Op twee punten deden zich problemen voor. Ten eerste was het, door het ontbreken van herschrijf-semantiek, niet mogelijk om de instructies in het geheugen van de in hoofdstuk 2 en 3 beschreven machines te laten zien. Ten tweede maakten het gebruikte typeringssysteem het noodzakelijk om enkele, overigens overbodige, constructoren te introduceren in de datastructuren die voor de beschrijving van vertalingen in hoofdstuk 4 en 5 werden gebruikt.

Curriculum vitae

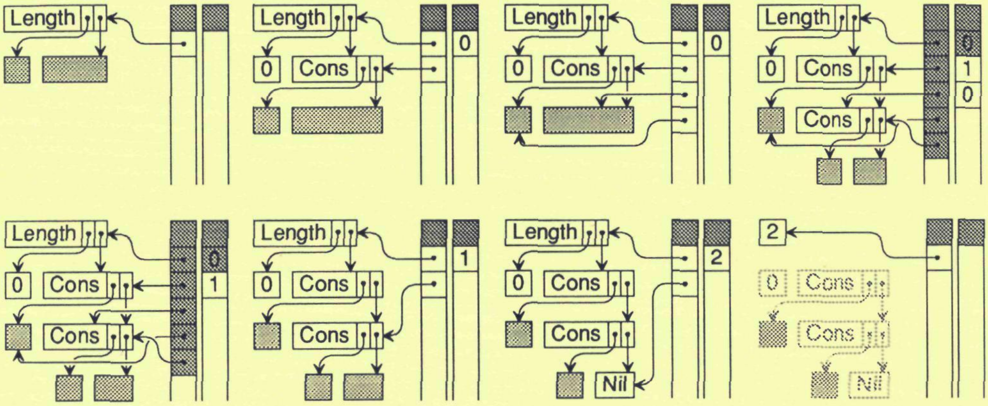
3 juli 1957 geboren te Nijmegen.

1970-1976 Atheneum-B aan het St. Odulphus Lyceum te Tilburg.

1976-1980 Kandidaats examen Natuurkunde (N4: hoofdvakken Natuurkunde en Scheikunde met bijvak Wiskunde) aan de Katholieke Universiteit Nijmegen (KUN).

1980-1983 Doctoraal examen Experimentele Natuurkunde met bijvakken capita uit de Natuurkunde en Informatica aan de KUN. Het afstudeerwerk stond onder begeleiding van Dr. P.I.M. Johannesma en betrof tweede-orde-signaalverwerking. Dit werk vond plaats op de afdeling Medische- en Biofysica onder leiding van Prof. Dr. J. Eggermont.

1984-1990 Wetenschapelijk medewerker/universitair docent/toegevoegd onderzoeker bij de sectie/discipline/vakgroep Informatica van de Faculteit Wiskunde en Natuurwetenschappen/Faculteit Wiskunde en Informatica aan de Katholieke Universiteit Nijmegen. Aanvankelijk was ik werkzaam bij de afdeling Machinearchitectuur en Bedrijfssystemen onder verantwoordelijkheid van Prof. Dr. Ir. R.T. Boute. Vervolgens heb ik gewerkt in de vakgroep Theoretische Informatica en Berekeningsmodellen bij Prof. Dr. H.P. Barendregt. Dit proefschrift is voltooid terwijl ik werkzaam was bij de afdeling Parallele Systemen en Berekeningsmodellen onder leiding van Dr. Ir. M.J. Plasmeijer.



ISBN 90-9003689-X